

ARCoSS

LNCS 14570

**Bernd Finkbeiner
Laura Kovács (Eds.)**

Tools and Algorithms for the Construction and Analysis of Systems

**30th International Conference, TACAS 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part I**

1
Part I



 **Springer**

OPEN ACCESS

Founding Editors

Gerhard Goos, Germany


Juris Hartmanis, USA

Editorial Board Members

Elisa Bertino, USA

Wen Gao, China

Bernhard Steffen , 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 <https://link.springer.com/bookseries/558>


Bernd Finkbeiner · Laura Kovács
Editors

Tools and Algorithms for the Construction and Analysis of Systems

30th International Conference, TACAS 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part I

Editors

Bernd Finkbeiner 
CISPA Helmholtz Center for Information
Security
Saarbrücken, Germany

Laura Kovács 
TU Wien
Vienna, Austria



ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-57245-6 ISBN 978-3-031-57246-3 (eBook)
<https://doi.org/10.1007/978-3-031-57246-3>

© The Editor(s) (if applicable) and The Author(s) 2024. 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

Paper in this product is recyclable.

ETAPS Foreword

Welcome to the 27th ETAPS! ETAPS 2024 took place in Luxembourg City, the beautiful capital of Luxembourg.

ETAPS 2024 is the 27th 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 languages, analysis tools, and formal approaches to software engineering. Organising 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 took place that attracted many researchers from all over the globe.

ETAPS 2024 received 352 submissions in total, 117 of which were accepted, yielding an overall acceptance rate of 33%. 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!

ETAPS 2024 featured the unifying invited speakers Sandrine Blazy (University of Rennes, France) and Lars Birkedal (Aarhus University, Denmark), and the invited speakers Ruzica Piskac (Yale University, USA) for TACAS and Jérôme Leroux (Laboratoire Bordelais de Recherche en Informatique, France) for FoSSaCS. Invited tutorials were provided by Tamar Sharon (Radboud University, the Netherlands) on computer ethics and David Monniaux (Verimag, France) on abstract interpretation.

As part of the programme we had the first ETAPS industry day. The goal of this day was to bring industrial practitioners into the heart of the research community and to catalyze the interaction between industry and academia. The day was organized by Nikolai Kosmatov (Thales Research and Technology, France) and Andrzej Wařowski (IT University of Copenhagen, Denmark).

ETAPS 2024 was organized by the SnT - Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg. The University of Luxembourg was founded in 2003. The university is one of the best and most international young universities with 6,000 students from 130 countries and 1,500 academics from all over the globe. The local organisation team consisted of Peter Y.A. Ryan (general chair), Peter B. Roenne (organisation chair), Maxime Cordy and Renzo Gaston Degiovanni (workshop chairs), Magali Martin and Isana Nascimento (event manager), Marjan Skrobot (publicity chair), and Afonso Arriaga (local proceedings chair). This team also

organised the online edition of ETAPS 2021, and now we are happy that they agreed to also organise a physical edition of ETAPS.

ETAPS 2024 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 Marieke Huisman (Twente, chair), Andrzej Wařowski (Copenhagen), Thomas Noll (Aachen), Jan Kofroň (Prague), Barbara König (Duisburg), Arnd Hartmanns (Twente), Caterina Urban (Inria), Jan Křetinský (Munich), Elizabeth Polgreen (Edinburgh), and Lenore Zuck (Chicago).

Other members of the steering committee are: Maurice ter Beek (Pisa), Dirk Beyer (Munich), Artur Boronat (Leicester), Luís Caires (Lisboa), Ana Cavalcanti (York), Ferruccio Damiani (Torino), Bernd Finkbeiner (Saarland), Gordon Fraser (Passau), Arie Gurfinkel (Waterloo), Reiner Hähnle (Darmstadt), Reiko Heckel (Leicester), Marijn Heule (Pittsburgh), Joost-Pieter Katoen (Aachen and Twente), Delia Kesner (Paris), Naoki Kobayashi (Tokyo), Fabrice Kordon (Paris), Laura Kovács (Vienna), Mark Lawford (Hamilton), Tiziana Margaria (Limerick), Claudio Menghi (Hamilton and Bergamo), Andrzej Murawski (Oxford), Laure Petrucci (Paris), Peter Y.A. Ryan (Luxembourg), Don Sannella (Edinburgh), Viktor Vafeiadis (Kaiserslautern), Stephanie Weirich (Pennsylvania), Anton Wijs (Eindhoven), and James Worrell (Oxford).

I would like to take this opportunity to thank all authors, keynote speakers, attendees, organizers of the satellite workshops, and Springer Nature for their support. ETAPS 2024 was also generously supported by a RESCOM grant from the Luxembourg National Research Foundation (project 18015543). I hope you all enjoyed ETAPS 2024.

Finally, a big thanks to both Peters, Magali and Isana and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

April 2024

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

This three-volume proceedings contains the papers presented at the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). TACAS 2024 was part of the 27th European Joint Conferences on Theory and Practice of Software (ETAPS 2024), which was held between April 6–11, 2024, in Luxembourg City, Luxembourg.

TACAS is a forum for researchers, developers and users interested in rigorous 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 2024 interleaves and integrates various disciplines, including formal verification of software and hardware systems, static analysis, probabilistic programming, program synthesis, concurrency, testing, simulations, verification of machine learning/autonomous systems, Cyber-Physical Systems, SAT/SMT solving, automated and interactive theorem proving, and proof checking.

There were four submission categories for TACAS 2024:

1. **Regular research papers** identifying and justifying a principled advance to the theoretical foundations for the construction and analysis of systems.
2. **Case study papers** describing the application of techniques developed by the community to a single problem or a set of problems of practical importance, preferably in a real-world setting.
3. **Regular tool papers** presenting a novel tool or a new version of an existing tool built using novel algorithmic and engineering techniques.
4. **Tool demonstration papers** demonstrating a new tool or application of an existing tool on a significant case-study.

Regular research, case study, and regular tool paper submissions were restricted to 16 pages, whereas tool demonstration papers to 6 pages, excluding the bibliography and appendices.

TACAS 2024 received 159 submissions, consisting of 114 regular research papers, 10 case study papers, 28 regular tool papers, and 7 tool demonstration papers. Each submission was assigned for review to at least three Program Committee (PC) members, who made use of subreviewers. Regular research papers were reviewed in double-blind mode, whereas case study, regular tool, and tool-demonstration papers were reviewed using a single-blind reviewing process.

Similarly to previous years, it was possible to submit an artifact alongside a paper. Artifact submission was mandatory for regular tool and tool demo papers, and voluntary for regular research and case study papers at TACAS 2024. An artifact might consist of a tool, models, proofs, or other data required for validation of the results of the paper. The Artifact Evaluation Committee (AEC) was tasked with reviewing the

artifacts, based on their documentation, ease of use, and, most importantly, whether the results presented in the corresponding paper could be accurately reproduced. Most of the evaluation was carried out using a standardized virtual machine to ensure consistency of the results, except for those artifacts that had special hardware or software requirements. Artifact evaluation at TACAS 2024 consisted of two rounds. The first round implemented the mandatory artifact evaluation of regular tool and tool demonstration papers; this round was carried out in parallel with the work of the PC. The judgment of the AEC was communicated to the PC and weighed in their discussion. The second round of artifact evaluation carried out the voluntary artifact evaluation of regular research and case study papers, and took place after paper acceptance notifications were sent out; authors of accepted regular research and case study papers were able to update and revise their respective artifacts before artifact evaluation started. In both rounds, the AEC provided 3 reviews per artifact and anonymously communicated with the authors to resolve apparent technical issues. In total, 104 artifacts were submitted and the AEC evaluated a total of 62 artifacts regarding their availability, functionality, and/or reusability. Papers with an artifact that were successfully evaluated include one or more badges on the first page, certifying the respective properties.

Selected papers were requested to provide a rebuttal in case a PC review gave rise to questions. Using the review reports and rebuttals, the PC had a thorough discussion on each paper. For regular tool and tool demonstration papers, the PC also discussed the corresponding artifact, using the AEC recommendations. As a result, the PC decided to accept 53 papers, out of which there were 35 regular research papers, 11 regular tool papers, 3 case study papers, and 4 tool demonstration papers. This corresponds to an overall acceptance rate of 33%. Each accepted paper at TACAS 2024 had either all positive reviews and/or a “championing” PC member who argued in favor of accepting the paper. All accepted papers at TACAS 2024 had a positive average review score.

TACAS 2024 also hosted SV-COMP 2024, the 13th International Competition on Software Verification. This event to compare tools evaluated 59 software systems for automatic verification of C and Java programs and 17 software systems for witness validation. The TACAS 2024 proceedings contains a competition report by the SV-Comp chair and organizer. From the 46 actively participating teams, the SV-Comp jury selected 16 short papers that describe the participating verification and validation systems. These 16 short papers are also published in the proceedings and were reviewed by a separate program committee (jury); each of these short papers was assessed by at least four jury members. Two sessions in the TACAS 2024 program were reserved for the presentation of the results: (1) a presentation session with a report by the competition chair and summaries by the developer teams of participating tools, and (2) an open community meeting in the second session.

We would like to thank everyone who helped to make TACAS 2024 successful. We thank the authors for submitting their papers to TACAS 2024. The PC members and additional reviewers did an excellent job in reviewing papers: they provided detailed reports and engaged in the PC discussions. We thank the TACAS steering committee, and especially its chair, Joost-Pieter Katoen, for his valuable advice. We are grateful to the ETAPS steering committee, and in particular its chair, Marieke Huisman, for supporting our changes and suggestions on the TACAS 2024 review process and final

program. We also acknowledge the invaluable support provided by the EasyChair developers. Lastly, we would like to thank the overall organization team of ETAPS 2024.

April 2024

Bernd Finkbeiner
Laura Kovács
PC Chairs

Hadar Frenkel
Michael Rawson
AEC Chairs

Dirk Beyer
SV-Comp Chair

Organization

Program Committee Chairs

Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Laura Kovács	TU Wien, Austria

Program Committee

Alessandro Abate	University of Oxford, UK
Erika Ábrahám	RWTH Aachen University, Germany
S. Akshay	IIT Bombay, India
Elvira Albert	Universidad Complutense de Madrid, Spain
Leonardo Alt	Ethereum Foundation
Suguman Bansal	Georgia Institute of Technology, USA
Nikolaj Bjørner	Microsoft Research, USA
Ahmed Bouajjani	IRIF, Université Paris Cité, France
Claudia Cauli	Amazon Web Services, UK
Rance Cleaveland	University of Maryland, USA
Mila Dalla Preda	University of Verona, Italy
Rayna Dimitrova	CISPA Helmholtz Center for Information Security, Germany
Madalina Erascu	West University of Timișoara, Romania
Javier Esparza	Technical University of Munich, Germany
Carlo A. Furia	USI - Università della Svizzera Italiana, Switzerland
Alberto Griggio	Fondazione Bruno Kessler, Italy
Arie Gurfinkel	University of Waterloo, Canada
Holger Hermanns	Saarland University, Germany
Marijn Heule	Carnegie Mellon University, USA
Hossein Hojjat	Tehran Institute for Advanced Studies, Iran
Nils Jansen	Ruhr-University Bochum, Germany and Radboud University, Netherlands
Sebastian Junges	Radboud University, Netherlands
Amir Kafshdar Goharshady	Hong Kong University of Science and Technology, China
Benjamin Lucien Kaminski	Saarland University, Germany and University College London, UK
Guy Katz	The Hebrew University of Jerusalem, Israel
Gergely Kovásznai	Eszterházy Károly University, Eger, Hungary
Tamás Kozsik	Eötvös Loránd University, Budapest, Hungary
Anthony Widjaja Lin	TU Kaiserslautern, Germany
Dorel Lucanu	Alexandru Ioan Cuza University, Romania

Filip Maric	University of Belgrade, Serbia
Laura Nenzi	University of Trieste, Italy
Aina Niemetz	Stanford University, USA
Elizabeth Polgreen	University of Edinburgh, UK
Kristin Yvonne Rozier	Iowa State University, USA
Cesar Sanchez	IMDEA Software Institute, Spain
Mark Santolucito	Barnard College, USA
Anne-Kathrin Schmuck	Max-Planck-Institute for Software Systems, Germany
Sharon Shoham	Tel Aviv University, Israel
Mihaela Sighireanu	University Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, France
Martin Suda	Czech Technical University in Prague, Czech Republic
Silvia Lizeth Tapia Tarifa	University of Oslo, Norway
Caterina Urban	Inria & ENS—PSL, France
Yakir Vizel	Technion, Israel
Tomas Vojnar	Brno University of Technology, Czech Republic
Georg Weissenbacher	TU Wien, Austria
Sarah Winkler	Free University of Bozen-Bolzano, Italy
Ningning Xie	University of Toronto and Google Brain, Canada

Artifact Evaluation Committee Chairs

Hadar Frenkel	CISPA Helmholtz Center for Information Security, Germany
Michael Rawson	TU Wien, Austria

Artifact Evaluation Committee

Tripti Agarwal	University of Utah, USA
Guy Amir	The Hebrew University of Jerusalem, Israel
Ahmed Bhayat	The University of Manchester, UK
Martin Blicha	University of Lugano, Switzerland
Alexander Bork	RWTH Aachen University, Germany
Lea Salome Brugger	ETH Zürich, Switzerland
Marco Campion	Inria & École Normale Supérieure—Université PSL, France
David Cerna	Czech Academy of Sciences Institute of Computer Science, Czech Republic
Kevin Cheang	Amazon Web Services, USA
Md Solimul Chowdhury	Carnegie Mellon University, USA
Vlad Craciun	BitDefender, UAIC, Romania
Jip J. Dekker	Monash University, Australia
Rafael Dewes	CISPA Helmholtz Center for Information Security, Germany
Oyendrila Dobe	Michigan State University, USA
Clemens Eisenhofer	TU Wien, Austria

Yizhak Elboher
 Raya Elsaleh
 Ferhat Erata
 Zafer Esen
 Aoyang Fang
 Pritam Gharat
 R. Govind
 Thomas Hader
 Philippe Heim

Maximilian Heisinger
 Alejandro
 Hernández-Cerezo
 Singh Hitarth

Petra Hozzová
 Jingmei Hu
 Tobias John
 Martin Jonáš
 Aniruddha Joshi
 Cezary Kaliszyk
 Elad Kinsbruner
 Åsmund Aqissiaq Arild
 Kløvstad
 Paul Kobialka
 Kerim Kochekov

Satoshi Kura
 Lorenz Leutgeb
 Marco Lewis
 Jing Liu
 Yonghui Liu
 Ioan Vlad Luca
 Kaushik Mallik
 Denis Mazzucato
 Baoluo Meng
 Niklas Metzger

Srinidhi Nagendra
 Jens Otten
 Jiří Pavela
 Bartosz Piotrowski
 Sumanth Prabhu
 Jyoti Prakash
 Siddharth Priya
 Felipe R. Monteiro

The Hebrew University of Jerusalem, Israel
 The Hebrew University of Jerusalem, Israel
 Yale University, USA
 Uppsala University, Sweden
 Chinese University of Hong Kong, Shenzhen, China
 Microsoft Research, India
 Uppsala University, Sweden
 TU Wien, Austria
 CISA Helmholtz Center for Information Security,
 Germany
 Johannes Kepler University Linz, Austria
 Complutense University of Madrid, Spain

 Hong Kong University of Science and Technology,
 China
 TU Wien, Austria
 Amazon, USA
 University of Oslo, Norway
 Masaryk University, Czech Republic
 UC Berkeley, USA
 University of Innsbruck, Austria
 Technion – Israel Institute of Technology, Israel
 University of Oslo, Norway

 University of Oslo, Norway
 Hong Kong University of Science and Technology,
 China
 National Institute of Informatics, Japan
 Max Planck Institute for Informatics, Germany
 Newcastle University, UK
 University of California, Irvine, USA
 Monash University, Australia
 West University of Timișoara, Romania
 Institute of Science and Technology Austria, Austria
 École Normale Supérieure, France
 GE Global Research, USA
 CISA Helmholtz Center for Information Security,
 Germany
 Chennai Mathematical Institute, India
 University of Oslo, Norway
 FIT VUT, Czech Republic
 IDEAS NCBR, Poland
 TRDDC, India
 University of Passau, Germany
 University of Waterloo, Canada
 Amazon Web Services, USA

Idan Refaeli	Hebrew University of Jerusalem, Israel
Simon Robillard	Université de Montpellier, France
Clara Rodríguez-Núñez	Complutense University of Madrid, Spain
Hans-Jörg Schurr	University of Iowa, USA
Tobias Seufert	University of Freiburg, Germany
Akshatha Shenoy	Tata Consultancy Services, India
Boris Shminke	Independent Researcher
Julian Siber	CISPA Helmholtz Center for Information Security, Germany
Cristian Simionescu	Alexandru Ioan Cuza University, Romania
Abhishek Kr Singh	Tel Aviv University, Israel
Alexander Steen	University of Greifswald, Germany
Geoff Sutcliffe	University of Miami, USA
Joseph Tafese	University of Waterloo, Canada
Jinhao Tan	University of Hong Kong, China
Abhishek Tiwari	University of Passau, Germany
Divyesh Unadkat	Synopsys, India
Lena Verscht	Saarland University and RWTH Aachen University, Germany
Christoph Wernhard	University of Potsdam, Germany
Haoze Wu	Stanford University, USA
Jiong Yang	National University of Singapore, Singapore
Yi Zhou	Carnegie Mellon University, USA

SV-COMP Program Committee and Jury

(more info: <https://sv-comp.sosy-lab.org/2024/committee.php>, sorted by tool name)

Dirk Beyer (Chair)	LMU Munich, Germany
Viktor Malík	Brno University of Technology, Czech Republic
Zhenbang Chen	National University of Defense Technology, China
Lei Bu	Nanjing University, China
Marek Chalupa	ISTA, Austria
Levente Bajczi	Budapest University of Technology and Economics, Hungary
Daniel Baier	LMU Munich, Germany
Thomas Lemberger	LMU Munich, Germany
Po-Chun Chien	LMU Munich, Germany
Hernán Ponce de León	Huawei Dresden Research Center, Germany
Fei He	Tsinghua University, China
Fatimah Aljaafari	University of Manchester, UK
Franz Brauße	University of Manchester, UK
Martin Spiessl	LMU Munich, Germany
Falk Howar	TU Dortmund, Germany
Simmo Saan	University of Tartu, Estonia

Hassan Mousavi	University of Tehran, Tehran Institute for Advanced Studies, Iran
Peter Schrammel	University of Sussex and Diffblue, UK
Zaiyu Cheng	University of Manchester, UK
Gidon Ernst	LMU Munich, Germany
Raphaël Monati	Inria and University of Lille, France
Jana (Philipp) Berger	RWTH Aachen, Germany
Veronika Šoková	Brno University of Technology, Czech Republic
Ravindra Metta	TCS, India
Vesal Vojdani	University of Tartu, Estonia
Nils Loose	University of Luebeck, Germany
Paulína Ayaziová	Masaryk University, Brno, Czech Republic
Martin Jonáš	Masaryk University, Brno, Czech Republic
Matthias Heizmann	University of Freiburg, Germany
Dominik Klumpp	University of Freiburg, Germany
Frank Schüssele	University of Freiburg, Germany
Daniel Dietsch	University of Freiburg, Germany
Priyanka Darke	Tata Consultancy Services, India
Marian Lingsch-Rosenfeld	LMU Munich, Germany

TACAS Steering Committee

Dirk Beyer	LMU Munich, Germany
Rance Cleaveland	University of Maryland, USA
Dana Fisman	Ben-Gurion University, Israel
Holger Hermanns	Universität des Saarlandes, Germany
Joost-Pieter Katoen (Chair)	RWTH Aachen, Germany and Universiteit Twente, Netherlands
Kim G. Larsen	Aalborg University, Denmark
Corina Păsăreanu	NASA Ames, USA

Additional Reviewers

Parosh Aziz Abdulla	Csaba Biró
Guy Amir	León Bohn
Andrei Arusoaie	Alberto Bombardelli
Shaun Azzopardi	Wael-Amine Boutglay
Thom Badings	Eline Bovy
Milan Banković	Matías Brizzio
Chinmayi Prabhu Baramashetru	Gianpiero Cabodi
Sebastien Bardin	Francesca Cairolì
Ludovico Battista	Marco Campion
Anna Becchi	Marco Carbone
Lena Becker	Martin Ceresa
Sidi Mohamed Beillahi	Kevin Cheang
Yoav Ben Shimon	Md Solimul Chowdhury

Alessandro Cimatti
Stefan Ciobaca
Cayden Codel
Jesús Correas
Arthur Correnson
Florin Craciun
Philipp Czerner
Tomáš Dacík
Luis Miguel Danielsson
Alessandro De Palma
Aldric Degorre
Rafael Dewes
Antonio Di Stasio
Denisa Diaconescu
Crystal Chang Din
Clemens Dubsloff
Serge Durand
Alec Edwards
Neta Elad
Yizhak Elboher
Raya Elsaleh
Constantin Enea
Zafer Esen
Soroush Farokhnia
Csaba Fazekas
Jan Fiedor
Emmanuel Fleury
James Fox
Felix Freiburger
Eden Frenkel
Florian Frohn
Maris Galesloot
Samir Genaim
Blaise Genest
Pamina Georgiou
Debarghya Ghoshdastidar
Adwait Godbole
Miguel Gomez-Zamalloa
Pablo Gordillo
Felipe Gorostiaga
R. Govind
Orna Grumberg
Roland Guttenberg
Serge Haddad
Philippe Heim
Martin Helfrich

Alejandro Hernández-Cerezo
Ivan Homoliak
Dániel Horpácsi
Karel Horák
Tzu-Han Hsu
Attila Házy
Miguel Isabel
Omri Isac
Radoslav Ivanov
Predrag Janicic
Chris Johannsen
Eduard Kamburjan
Ambrus Kaposi
Joost-Pieter Katoen
Lutz Klinkenberg
Paul Kobialka
Wietze Koops
Katherine Kosaian
David Kozák
Merlijn Krale
Valentin Krasotin
Loes Kruger
Gabor Kusper
Maximilian Alexander Köhl
Faezeh Labbaf
Nham Le
Matthieu Lemerre
Ondrej Lengal
Dániel Lukács
Michael Luttenberger
Viktor Malík
Alessio Mansutti
Niccolò Marastoni
Oliver Markgraf
Enrique Martin-Martin
Ruben Martins
Denis Mazzucato
Tobias Meggendorfer
Roland Meyer
Marcel Moosbrugger
Federico Mora
Alexander Nadel
Satya Prakash Nayak
Tobias Nießen
Andres Noetzli
Mohammed Nsaif

Robin Ohs
Emanuel Onica
Michele Pasqua
Andrea Pferscher
Zoltan Porkolab
Kostiantyn Potomkin
Mathias Preiner
Siddharth Priya
Tim Quatmann
Peter Rakya
Omer Rappoport
Jakob Rath
Rodrigo Raya
Adrian Rebola Pardo
Gianluca Redondi
Joseph Reeves
Luke Rickard
Andoni Rodriguez
Clara Rodríguez-Núñez
Adam Rogalewicz
Enrique Román Calvo
Guillermo Román-Díez
Vlad Rusu
Krishna S.
Irmak Saglam
Matteo Sammartino
Raimundo Saona Urmeneta
Gaia Saveri
Andre Schidler
Christoph Schmidl
Andreas Schmidt
Yannik Schnitzer

Philipp Schröer
Stefan Schwoon
Traian Florin Serbanuta
Daqian Shao
Xujie Si
Mate Soos
Martin Steffen
Gregory Stock
Sana Stojanović-Đurđević
Bernardo Subercaseaux
Marnix Suilen
Mantas Šimkus
Máté Tejfel
Simon Thompson
Hazem Torfah
Dmitriy Traytel
Marck van der Vegt
Sarat Varanasi
Sarat Chandra Varanasi
Ennio Visconti
Sebastian Wolff
Yechuan Xia
Mitsuharu Yamamoto
Raz Yerushalmi
Emre Yolcu
Pian Yu
Hanwei Zhang
Zhiwei Zhang
Shufang Zhu
Djordje Zikelic
Zoltán Zimborás
Dominic Zimmer

Contents – Part I

SAT and SMT Solving

DRAT Proofs of Unsatisfiability for SAT Modulo Monotonic Theories	3
<i>Nick Feng, Alan J. Hu, Sam Bayless, Syed M. Iqbal, Patrick Trentin, Mike Whalen, Lee Pike, and John Backes</i>	
Z3-NOODLER: An Automata-based String Solver	24
<i>Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč</i>	
TaSSAT: Transfer and Share SAT	34
<i>Md Solimul Chowdhury, Cayden R. Codel, and Marijn J. H. Heule</i>	
Speculative SAT Modulo SAT	43
<i>V. K. Hari Govind, Isabel Garcia-Contreras, Sharon Shoham, and Arie Gurfinkel</i>	
Happy Ending: An Empty Hexagon in Every Set of 30 Points	61
<i>Marijn J. H. Heule and Manfred Scheucher</i>	

Synthesis

Fully Generalized Reactivity(1) Synthesis.	83
<i>Rüdiger Ehlers and Ayrat Khalimov</i>	
Knor: reactive synthesis using Oink.	103
<i>Tom van Dijk, Feije van Abbema, and Naum Tomov</i>	
On Dependent Variables in Reactive Synthesis	123
<i>S. Akshay, Eliyahu Basa, Supratik Chakraborty, and Dror Fried</i>	
CESAR: Control Envelope Synthesis via Angelic Refinements	144
<i>Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and André Platzer</i>	

Logic and Decidability

Answering Temporal Conjunctive Queries over Description Logic	
Ontologies for Situation Recognition in Complex Operational Domains	167
<i>Lukas Westhofen, Christian Neurohr, Jean Christoph Jung, and Daniel Neider</i>	

Deciding Boolean Separation Logic via Small Models	188
<i>Tomáš Dacík, Adam Rogalewicz, Tomáš Vojnar, and Florian Zuleger</i>	

Asynchronous Subtyping by Trace Relaxation	207
<i>Laura Bocchi, Andy King, and Maurizio Murgia</i>	

Program Analysis and Proofs

SootUp: A Redesign of the Soot Static Analysis Framework	229
<i>Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He</i>	

Formally verified asymptotic consensus in robust networks	248
<i>Mohit Tekriwal, Avi Tachna-Fram, Jean-Baptiste Jeannin, Manos Kapritsos, and Dimitra Panagou</i>	

Formally Verifying an Efficient Sorter	268
<i>Bernhard Beckert, Peter Sanders, Mattias Ulbrich, Julian Wiesler, and Sascha Witt</i>	

Explainable Online Monitoring of Metric First-Order Temporal Logic	288
<i>Leonardo Lima, Jonathan Julián Huerta y Munive, and Dmitriy Traytel</i>	

Proof Checking

IsaRARE: Automatic Verification of SMT Rewrites in Isabelle/HOL	311
<i>Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli</i>	

Automate where Automation Fails: Proof Strategies for Frama-C/WP	331
<i>Loïc Correnson, Allan Blanchard, Adel Djoudi, and Nikolai Kosmatov</i>	

VeSCMul: Verified Implementation of S-C-Rewriting for Multiplier Verification	340
<i>Mertcan Temel</i>	

A Logical Treatment of Finite Automata	350
<i>Nishant Rodrigues, Mircea Octavian Sebe, Xiaohong Chen, and Grigore Roşu</i>	

A State-of-the-Art Karp-Miller Algorithm Certified in Coq	370
<i>Thibault Hilaire, David Ilcinkas, and Jérôme Leroux</i>	

Author Index	391
------------------------	-----

Contents – Part II

Model Checking

JPF: From 2003 to 2023	3
<i>Cyrille Artho, Pavel Parízek, Daohan Qu, Varadraj Galgali, and Pu (Luke) Yi</i>	

Hitching a Ride to a Lasso: Massively Parallel On-The-Fly LTL Model Checking	23
<i>Muhammad Osama and Anton Wijs</i>	

Towards Safe Autonomous Driving: Model Checking a Behavior Planner during Development	44
<i>Lukas König, Christian Heinzemann, Alberto Griggio, Michaela Klauck, Alessandro Cimatti, Franziska Henze, Stefano Tonetta, Stefan Küperkoch, Dennis Fassbender, and Michael Hanselmann</i>	

Enhancing GenMC’s Usability and Performance	66
<i>Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis</i>	

Automata and Learning

Scalable Tree-based Register Automata Learning	87
<i>Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist</i>	

Small Test Suites for Active Automata Learning	109
<i>Loes Kruger, Sebastian Junges, and Jurriaan Rot</i>	

MATA: A Fast and Simple Finite Automata Library	130
<i>David Chocholatý, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík, Martin Hruška, Ondřej Lengál, and Juraj Síc</i>	

Software Verification

Accelerated Bounded Model Checking Using Interpolation Based Summaries	155
<i>Mayank Solanki, Prantik Chatterjee, Akash Lal, and Subhajit Roy</i>	

Weakest Precondition Inference for Non-Deterministic Linear Array Programs	175
<i>Sumanth Prabhu S, Deepak D'Souza, Supratik Chakraborty, R Venkatesh, and Grigory Fedyukovich</i>	
Automated Software Verification of Hyperliveness	196
<i>Raven Beutner</i>	
A Comprehensive Specification and Verification of the L4 Microkernel API	217
<i>Leping Zhang, Yongwang Zhao, and Jianxin Li</i>	
Probabilistic Systems	
Accurately Computing Expected Visiting Times and Stationary Distributions in Markov Chains	237
<i>Hannah Mertens, Joost-Pieter Katoen, Tim Quatmann, and Tobias Winkler</i>	
CTMCs with Imprecisely Timed Observations	258
<i>Thom Badings, Matthias Volk, Sebastian Junges, Marielle Stoelinga, and Nils Jansen</i>	
Pareto Curves for Compositionally Model Checking String Diagrams of MDPs	279
<i>Kazuki Watanabe, Marck van der Vegt, Ichiro Hasuo, Jurriaan Rot, and Sebastian Junges</i>	
Learning Explainable and Better Performing Representations of POMDP Strategies	299
<i>Alexander Bork, Debraj Chakraborty, Kush Grover, Jan Křetínský, and Stefanie Mohr</i>	
Simulations	
Dissipative quadratizations of polynomial ODE systems.	323
<i>Yubo Cai and Gleb Pogudin</i>	
Forward and Backward Constrained Bisimulations for Quantum Circuits	343
<i>A. Jiménez-Pastor, K. G. Larsen, M. Tribastone, and M. Tschaikowski</i>	

A Parallel and Distributed Quantum SAT Solver Based on Entanglement
and Teleportation 363
 *Shang-Wei Lin, Tzu-Fan Wang, Yean-Ru Chen, Zhe Hou, David Sanán,
 and Yon Shin Teo*

Author Index 383

Contents – Part III

Neural Networks

Provable Preimage Under-Approximation for Neural Networks	3
<i>Xiyue Zhang, Benjie Wang, and Marta Kwiatkowska</i>	
Training for Verification: Increasing Neuron Stability to Scale DNN Verification	24
<i>Dong Xu, Nusrat Jahan Mozumder, Hai Duong, and Matthew B. Dwyer</i>	
NeuroSynt: A Neuro-symbolic Portfolio Solver for Reactive Synthesis	45
<i>Matthias Cosler, Christopher Hahn, Ayham Omar, and Frederik Schmitt</i>	

Testing and Verification

HALIVER: Deductive Verification and Scheduling Languages Join Forces	71
<i>Lars B. van den Haak, Anton Wijs, Marieke Huisman, and Mark van den Brand</i>	
Gray-Box Fuzzing via Gradient Descent and Boolean Expression Coverage . . .	90
<i>Martin Jonáš, Jan Strejček, Marek Trtik, and Lukáš Urban</i>	
Fast Symbolic Computation of Bottom SCCs	110
<i>Anna Blume Jakobsen, Rasmus Skibdahl Melanchton Jørgensen, Jaco van de Pol, and Andreas Pavlogiannis</i>	
Btor2-Cert: A Certifying Hardware-Verification Framework Using Software Analyzers	129
<i>Zsófia Ádám, Dirk Beyer, Po-Chun Chien, Nian-Ze Lee, and Nils Sirrenberg</i>	

Games

Auction-Based Scheduling	153
<i>Guy Avni, Kaushik Mallik, and Suman Sadhukhan</i>	
Most General Winning Secure Equilibria Synthesis in Graph Games	173
<i>Satya Prakash Nayak and Anne-Kathrin Schmuck</i>	

On-The-Fly Algorithm for Reachability in Parametric Timed Games	194
<i>Mikael Bisgaard Dahlsen-Jensen, Baptiste Fievet, Laure Petrucci, and Jaco van de Pol</i>	
Rabin Games and Colourful Universal Trees	213
<i>Rupak Majumdar, Irmak Sağlam, and K. S. Thejaswini</i>	
Concurrency	
Decidable Verification under Localized Release-Acquire Concurrency	235
<i>Abhishek Kr Singh and Ori Lahav</i>	
OxiDD: A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust.	255
<i>Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl</i>	
Verification under TSO with an infinite Data Domain	276
<i>Parosh Aziz Abdulla, Mohamed Faouzi Atig, Florian Furbach, and Shashwat Garg</i>	
13th Competition on Software Verification—SV-Comp 2024	
State of the Art in Software Verification and Witness Validation: SV-COMP 2024	299
<i>Dirk Beyer</i>	
ConcurrentWitness2Test: Test-Harnessing the Power of Concurrency (Competition Contribution).	330
<i>Levente Bajczi, Zsófia Ádám, and Zoltán Micskei</i>	
GOBLINT VALIDATOR: Correctness Witness Validation by Abstract Interpretation (Competition Contribution).	335
<i>Simmo Saan, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl</i>	
WITCH 3: Validation of Violation Witnesses in the Witness Format 2.0 (Competition Contribution).	341
<i>Paulína Ayaziová and Jan Strejček</i>	
AISE: A Symbolic Verifier by Synergizing Abstract Interpretation and Symbolic Execution (Competition Contribution)	347
<i>Zhen Wang and Zhenbang Chen</i>	

BUBAAK-SpLit: Split what you cannot verify (Competition contribution)	353
<i>Marek Chalupa and Cedric Richter</i>	
CPACHECKER 2.3 with Strategy Selection (Competition Contribution)	359
<i>Daniel Baier, Dirk Beyer, Po-Chun Chien, Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Marian Lingsch-Rosenfeld, Martin Spiessl, Henrik Wachowitz, and Philipp Wendler</i>	
CPV: A Circuit-Based Program Verifier (Competition Contribution)	365
<i>Po-Chun Chien and Nian-Ze Lee</i>	
EmergenTheta: Verification Beyond Abstraction Refinement (Competition Contribution)	371
<i>Levente Bajczi, Dániel Szekeres, Milán Mondok, Zsófia Ádám, Márk Somorjai, Csanád Telbisz, Mihály Dobos-Kovács, and Vince Molnár</i>	
ESBMC v7.4: Harnessing the Power of Intervals (Competition Contribution)	376
<i>Rafael Sá Menezes, Mohannad Aldughaim, Bruno Farias, Xianzhiyu Li, Edoardo Manino, Fedor Shmarov, Kunjian Song, Franz Brauße, Mikhail R. Gadelha, Norbert Tihanyi, Konstantin Korovin, and Lucas C. Cordeiro</i>	
GOBLINT: Abstract Interpretation for Memory Safety and Termination (Competition Contribution)	381
<i>Simmo Saan, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl</i>	
Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)	387
<i>Raphaël Monat, Marco Milanese, Francesco Parolini, Jérôme Boillot, Abdelraouf Ouadjaout, and Antoine Miné</i>	
PROTON: PRObes for Termination Or Not (Competition Contribution)	393
<i>Ravindra Metta, Hrishikesh Karmarkar, Kumar Madhukar, R. Venkatesh, and Supratik Chakraborty</i>	
SWAT: Modular Dynamic Symbolic Execution for Java Applications using Dynamic Instrumentation (Competition Contribution)	399
<i>Nils Loose, Felix Mächtle, Florian Sieck, and Thomas Eisenbarth</i>	

Symbiotic 10: Lazy Memory Initialization and Compact Symbolic
Execution (Competition Contribution) 406
*Martin Jonáš, Kristián Kumor, Jakub Novák, Jindřich Sedláček,
Marek Trtík, Lukáš Zaoral, Paulína Ayaziová, and Jan Strejček*

Theta: Abstraction Based Techniques for Verifying Concurrency
(Competition Contribution). 412
*Levente Bajczi, Csanád Telbisz, Márk Somorjai, Zsófia Ádám,
Mihály Dobos-Kovács, Dániel Szekeres, Milán Mondok,
and Vince Molnár*

Ultimate Automizer and the Abstraction of Bitwise Operations
(Competition Contribution). 418
*Frank Schüssele, Manuel Bentele, Daniel Dietsch, Matthias Heizmann,
Xinyu Jiang, Dominik Klumpp, and Andreas Podelski*

Author Index 425

SAT and SMT Solving



DRAT Proofs of Unsatisfiability for SAT Modulo Monotonic Theories

Nick Feng^{1(✉)}, Alan J. Hu², Sam Bayless³, Syed M. Iqbal³, Patrick Trentin³, Mike Whalen³, Lee Pike³, and John Backes³

¹ Dept. of Computer Science, University of Toronto, Toronto, Canada
fengnick@cs.toronto.edu

² Dept. of Computer Science, University of British Columbia, Vancouver, Canada
ajh@cs.ubc.ca

³ Amazon Web Services, Seattle, Minneapolis, Portland, USA
{sabayles,iqsy,trentinp,mww,leepike,jbackes}@amazon.com

Abstract. Generating proofs of unsatisfiability is a valuable capability of most SAT solvers, and is an active area of research for SMT solvers. This paper introduces the first method to efficiently generate proofs of unsatisfiability specifically for an important subset of SMT: SAT Modulo *Monotonic Theories* (SMMT), which includes many useful finite-domain theories (e.g., bit vectors and many graph-theoretic properties) and is used in production at Amazon Web Services. Our method uses propositional definitions of the theory predicates, from which it generates compact Horn approximations of the definitions, which lead to efficient DRAT proofs, leveraging the large investment the SAT community has made in DRAT. In experiments on practical SMMT problems, our proof generation overhead is minimal (7.41% geometric mean slowdown, 28.8% worst-case), and we can generate and check proofs for many problems that were previously intractable.

An extended version of this paper, which includes appendices with proofs and additional results, is available at <https://doi.org/10.48550/arXiv.2401.10703>

1 Introduction

This paper introduces the first method to efficiently generate and check proofs of unsatisfiability for SAT Modulo Monotonic Theories (SMMT), an important fragment of general SMT. The motivation for this work rests on these premises:

- *Proofs of UNSAT are valuable, for propositional SAT as well as SMT.* Obviously, an independently checkable proof increases trust, which is important because an incorrect UNSAT result can result in certifying correctness of an incorrect system. Additionally, proofs are useful for computing abstractions [30,17,25] via interpolation in many application domains including model checking [30] and software analysis [29,23].

- *SMMT is a worthy fragment of SMT as a research target.* SMMT [9] is a technique for efficiently supporting finite, monotonic theories in SMT solvers. E.g., reachability in a graph is monotonic in the sense that adding edges to the graph only increases reachability, and an example SMMT query would be whether there exists a configuration of edges such that node a can reach node b , but node c can't reach node d . (More formal background on SMMT is in Sec. 2.2.) The most used SMMT theories are graph reachability and max-flow, along with bit-vector addition and comparison. Applications include circuit escape routing [11], CTL synthesis [28], virtual data center allocation [12], and cloud network security and debugging [2,8], with the last two applications being deployed in production by Amazon Web Services (AWS). Indeed, our research was specifically driven by industrial demand.
- *DRAT is a desirable proof format.* (Here, we include related formats like DRUP [27], GRIT [19], and LRAT [18]. DRAT is explained in Sec. 2.1.) For an independent assurance of correctness, the proof *checker* is the critical, trusted component, and hence must be as trustworthy as possible. For (propositional) SAT, the community has coalesced around the DRAT proof format [37], for which there exist independent, efficient proof checkers [37], mechanically verified proof checkers [38], and even combinations that are fast as well as mechanically proven [18]. The ability to emit DRAT proof certificates has been required for solvers in the annual SAT Competition since 2014.

Unfortunately, DRAT is propositional, so general SMT solvers need additional mechanisms to handle theory reasoning [6]. For example, Z3 [32] outputs natural-deduction-style proofs [31], which can be reconstructed inside the interactive theorem prover Isabelle/HOL [14,15]. Similarly, veriT [16] produces resolution proof traces with theory lemmas, and supports proof reconstruction in both Coq [1] and Isabelle [21,5,4]. As a more general approach, CVC4 [7] produces proofs in the LFSC format [36], which is a meta-logic that allows describing theory-specific proof rules for different SMT theories. Nevertheless, given the virtues of DRAT, SMT solvers have started to harness it for the propositional reasoning, e.g., CVC4 supports DRAT proofs for bit-blasting of the bit-vector theory, which are then translated into LFSC [34], and Otoni et al. [33] propose a DRAT-based proof certificate format for propositional reasoning that they extend with theory-specific certificates. However, in both cases, the final proof certificate is not purely DRAT, and any theory lemmas must be checked by theory-specific certificate checkers.

- *For typical finite-domain theories, defining theory predicates propositionally is relatively straightforward.* The skills to design and implement theory-specific proof systems are specialized and not widely taught. In contrast, if we treat a theory predicate as simply a Boolean function, then anyone with basic digital design skills can build a circuit to compute the predicate (possibly using readily available commercial tools) and then apply the Tseitin transform to convert the circuit to CNF. (This is known as “bit-blasting”, but we will see later that conventional bit-blasting is too inefficient for SMMT.)

From a practical, user-level perspective, the contribution of this paper is the first efficient proof-generating method for SMMT. Our method scales to industrial-size instances and generates pure DRAT proofs.

From a theoretical perspective, the following contributions underlie our method:

- We introduce the notion of one-sided propositional definitions for refutation proof. Having different definitions for a predicate vs. its complement allows for more compact and efficient constructions.
- We show that SMMT theories expressed in Horn theory enable linear-time (in the size of the Horn definition) theory lemma checking via reverse unit propagation (RUP), and hence DRAT.
- We propose an on-the-fly transformation that uses hints from the SMMT solver to over-approximate any CNF encoding of a monotonic theory predicate into a linear-size Horn upper-bound, and prove that the Horn upper-bound is sufficient for checking theory lemmas in any given proof via RUP.
- We present efficient, practical propositional definitions for the main monotonic theories used in practice: bit-vector summation and comparison, and reachability and max-flow on symbolic graphs.

(As an additional minor contribution, we adapt the BackwardCheck procedure from DRAT-Trim [27] for use with SMT, and evaluate its effectiveness in our proof checker.)

We implemented our method in the MonoSAT SMMT solver [10]. For evaluation, we use two sets of benchmarks derived from practical, industrial problems: multilayer escape routing [11], and cloud network reachability [2].⁴ Our results show minimal runtime overhead on the solver (geometric mean slowdown 7.4%, worst-case 28.8% in our experiments), and we generate and check proofs for many problem instances that are otherwise intractable.

2 Background

2.1 Propositional SAT and DRAT

We assume the reader is familiar with standard propositional satisfiability on CNF. Some notational conventions in our paper are: we use lowercase letters for literals and uppercase letters for clauses (or other sets of literals); for a literal x , we denote the variable of x by $var(x)$; we will interchangeably treat an *assignment* either as a mapping of variables to truth values \top (true) or \perp (false), or as a set of non-conflicting (i.e., does not contain both x and its complement \bar{x}) literals, with positive (negative) literals for variables assigned \top (\perp); assignments can be *total* (assigns truth values to every variable) or *partial* (some variables unassigned); and given a formula F and assignment M , we use the vertical bar $F|_M$ to denote reducing the formula by the assignment, i.e., discarding falsified

⁴ Available at <https://github.com/NickF0211/MonoProof>.

literals from clauses and satisfied clauses from the formula. (An empty clause denotes \perp ; an empty formula, \top .)

This paper focuses on proofs of unsatisfiability. In proving a formula F UNSAT, a clause C is *redundant* if F and $F \wedge C$ are equisatisfiable [26]. A proof of unsatisfiability is simply a sequence of redundant clauses culminating in \perp , but where the redundancy of each clause can be easily checked. However, checking redundancy is coNP-hard. A clause that is *implied* by F , which we denote by $F \models C$, is guaranteed redundant, and we can check implication by checking the unsatisfiability of $F \wedge \overline{C}$, but this is still coNP-complete. Hence, proofs use restricted proof rules that guarantee redundancy. For example, the first automated proofs of UNSAT used resolution to generate implied clauses, until implying \perp by resolving a literal l with its complement \bar{l} [20,39]. In practice, however, resolution proofs grow too large on industrial-scale problems.

DRAT [37] is a much more compact and efficient system for proving unsatisfiability. It is based on *reverse unit propagation* (RUP), which we explain here.⁵ A *unit clause* is a clause containing one literal. If L is the set of literals appearing in the unit clauses of a formula F , the *unit clause rule* computes $F|_L$, and the repeated application of the unit clause rule until a fixpoint is called *unit propagation* (aka *Boolean constraint propagation*). Given a clause C , its negation \overline{C} is a set of unit clauses, and we denote by $F \vdash_1 C$ if $F \wedge \overline{C}$ derives a conflict through unit propagation. Notice that $F \vdash_1 C$ implies $F \models C$, but is computationally easy to check. The key insight [24] behind RUP is that modern CDCL SAT solvers make progress by deriving learned clauses, whose redundancy is, by construction, checkable via unit propagation. Proof generation, therefore, is essentially just logging the sequence of learned clauses leading to \perp , and proof checking is efficiently checking \vdash_1 of the relevant learned clauses.

2.2 SAT Modular Monotonic Theories (SMMT)

We define a Boolean *positive monotonic predicate* as follows:

Definition 1 (Positive Monotonic Predicate). A predicate $p : \{0,1\}^n \rightarrow \{0,1\}$ is *positively monotonic with respect to the input a_i* iff

$$p(a_1, \dots, a_{i-1}, 0, a_{i+1}, \dots) \implies p(a_1, \dots, a_{i-1}, 1, a_{i+1}, \dots)$$

The predicate p is a *positive monotonic predicate* iff p is *positively monotonic with respect to every input*.

Negative monotonic predicates are defined analogously. If a predicate p is positively monotonic w.r.t. some inputs A^+ and negatively monotonic w.r.t. the rest of inputs A^- , it is always possible to rewrite the predicate as a positive monotonic predicate p' over input A^+ and $\{\bar{a} \mid a \in A^-\}$. For ease of exposition,

⁵ RUP is all we use in this paper. RAT is a superset of RUP, by essentially doing one step of resolution as a “lookahead” before checking RUP of the resolvents. The “D” in DRAT stands for “deletion”, meaning the proof format also records clause deletions.

and without loss of generality, we will describe our theoretical results assuming positive monotonic predicates only (except where noted otherwise).

Given a monotonic predicate p over input A , we will use boldface \mathbf{p} as the *predicate atom* for p , i.e., the predicate atom is a Boolean variable in the CNF encoding of the theory, indicating whether $p(A)$ is true or not. The *theory of \mathbf{p}* is the set of valid implications in the form of $\mathcal{M}_A \Rightarrow \mathbf{p}$ where \mathcal{M}_A is a partial assignment over A .

The following are the most used monotonic theories:

Graph Reachability: Given a graph $G = (V, E)$, where V and E are sets of vertices and edges, the graph reachability theory contains the reachability predicates $reach_u^v$ on the input variables $e_1, e_2 \dots e_m \in E$, where $u, v \in V$. The predicate holds iff node u can reach v in the graph G by using only the subset of edges whose corresponding variable e_i is true. The predicate is positively monotonic because enabling more edges will not make reachable nodes unreachable, and disabling edge will not make unreachable nodes reachable.

Bit-Vector Summation and Comparison: Given two bit-vectors (BV) \vec{a} and \vec{b} , the theory of BV comparison contains the predicate $\vec{a} \geq \vec{b}$, whose inputs are the bits of \vec{a} and \vec{b} . The predicate holds iff the value (interpreted as an integer) of \vec{a} is greater or equal to the value of \vec{b} . The predicate is positively monotonic for the variables of \vec{a} and negatively monotonic for the variables of \vec{b} , because changing any 0 to a 1 in \vec{a} makes it bigger, and changing any 1 to 0 in \vec{b} makes it smaller. Similarly, given two sets of BVs \vec{A} and \vec{B} , the theory of comparison between sums contains the predicate $\sum \vec{A} \geq \sum \vec{B}$ whose inputs are the boolean variables from all BVs in \vec{A} and \vec{B} . The predicate holds iff the sum of the BVs in \vec{A} is greater or equal to the sum of the BVs in \vec{B} , and is positively monotonic in \vec{A} and negatively monotonic in \vec{B} .

S-T Max Flow Given a graph $G = (V, E)$, for every edge $e \in E$, let its *capacity* be represented by the BV $c\vec{a}p_e$. For two vertices $s, t \in V$, and a BV \vec{z} , the max-flow theory contains the predicates $MF_s^t \geq \vec{z}$ over the input variables $e_1, e_2 \dots e_n \in E$ and $c\vec{a}p_{e_1}, c\vec{a}p_{e_2} \dots c\vec{a}p_{e_n}$. The predicate holds iff the maximum flow from the source s to the target t is greater or equal to \vec{z} , using only the enabled edges (as in the reachability theory) with their specified capacities.

The SMMT Framework [10] describes how to extend a SAT or SMT solver with Boolean monotonic theories. The framework has been implemented in the SMT solver MonoSAT, which has been deployed in production by Amazon Web Services to reason about a wide range of network properties [2,8]. The framework performs theory propagation and clause learning for SMMT theories as follows: (In this description, we use P for the set of positive monotonic predicates, and S for the set of Boolean variables that are arguments to the predicates.)

Theory Propagation: Given a partial assignment M , let M_s be the partial assignment over S . The SMMT framework forms two complete assignments

of M_s : one with all unassigned s atoms assigned to false (M_s^-), one with all unassigned s atoms assigned to true (M_s^+). Since M_s^- and M_s^+ are each complete assignments of S , they can be used to determine the value of P atoms. Since every $\mathbf{p} \in P$ is positively monotonic, (1) if $M_s^- \Rightarrow \mathbf{p}$, then $M_s \Rightarrow \mathbf{p}$, and (2) if $M_s^+ \Rightarrow \neg \mathbf{p}$, then $M_s \Rightarrow \neg \mathbf{p}$. The framework uses M_s^- and M_s^+ as the under- and over-approximation for theory propagation over P atoms. Moreover, the framework attaches $M_s \Rightarrow \mathbf{p}$ or $M_s \Rightarrow \neg \mathbf{p}$ as the reason clause for the theory propagation.

Clause Learning: For some predicates, a witness can be efficiently generated during theory propagation, as a sufficient condition to imply the predicate p . For example, in graph reachability, suppose $M_s^- \Rightarrow \text{reach}_{u,v,G}$ for a given under-approximation M_s^- . Standard reachability algorithms can efficiently find a set of edges $M'_s \subseteq M_s$ that forms a path from u to v . When such a witness is available, instead of learning $M_s \Rightarrow \mathbf{p}$, the framework would use the path witness to learn the stronger clause $M'_s \Rightarrow \mathbf{p}$. Witness-based clause learning is theory specific (and implementation specific); if a witness is not available or cannot be efficiently generated in practice for a particular predicate, the framework will learn the weaker clause $M_s \Rightarrow \mathbf{p}$.

3 Overview of Our Method

Most leading SMT solvers, including MonoSAT, use the DPLL(T) framework [22], in which a CDCL propositional SAT solver coordinates one or more theory-specific solvers. A DPLL(T) solver behaves similarly to a CDCL propositional SAT solver — making decisions, performing unit propagation, analyzing conflicts, learning conflict clauses — except that the theory solvers will also introduce new clauses (i.e., theory lemmas) into the clause database, which were derived via theory reasoning, and whose correctness relies on the semantics of the underlying SMT theory. These theory lemmas cannot (in general) be derived from the initial clause database, and so cannot be verified using DRAT. Therefore, the problem of producing a proof of UNSAT in SMT reduces to the problem of proving the theory lemmas.

A direct approach would be to have the SMT solver emit a partial DRAT proof certificate, in which each theory lemma is treated as an axiom. This partial proof is DRAT-checkable, but each theory lemma becomes a new proof obligation. The theory lemmas could subsequently be verified using external (non-DRAT), trusted, theory-specific proof-checking procedures. This is the approach recently proposed by Otoni et al. [33].

We take such an approach as a starting point, but instead of theory-specific proof procedures, we use propositional definitions of the theory semantics to add clauses sufficient to prove (by RUP) the theory lemmas. The resulting proof is purely DRAT, checkable via standard DRAT checkers, with no theory-specific proof rules. Fig. 1 explains our approach in more detail; Sec. 4 dives into how we derive the added clauses; and Sec. 5 gives sample propositional theory definitions.

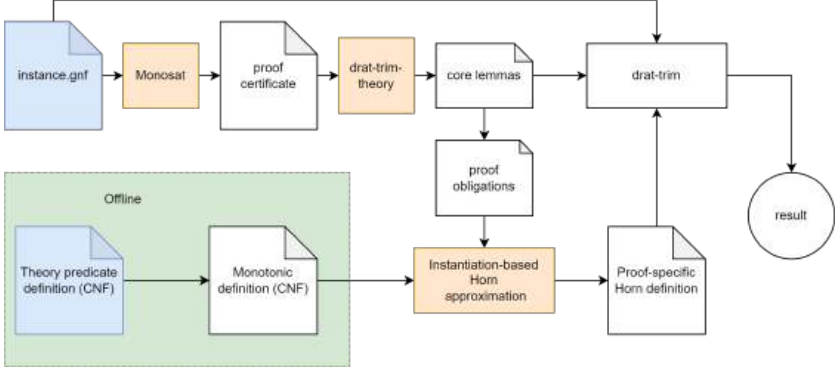


Fig. 1. Overview of Our Proof Generation and Checking Method. Inputs (the problem instance file and the propositional definitions of theory predicates) are colored blue; new and modified components are colored orange. Starting from the top-left is the SMMT problem instance, which is solved by MonoSAT. We extended MonoSAT to emit a DRAT-style proof certificate, consisting of learned (via propositional or theory reasoning) clauses, similar to what is proposed in [33]. The proof certificate is optionally pre-processed by *drat-trim-theory*, in which we modified the BackwardCheck procedure [27] to perform a backward traversal from the final \perp , outputting a subset of lemmas sufficient (combined with the original clause database) to derive \perp . This is extra work (since a full BackwardCheck is later performed by unmodified *drat-trim* for the final proof verification at the top-right of the figure), but allows us to avoid verifying some theory lemmas that are not relevant to the final proof. The resulting core lemmas are split between the propositional learned clauses, which go straight (right) to *drat-trim*, and the theory learned clauses, which are our proof obligations. The heart of our method is the instantiation-based Horn approximation (bottom-center, described in Sec. 4). In this step, we use the proof obligations as hints to transform the pre-defined, propositional theory definitions (bottom-left, examples in Sec. 5) into proof-specific Horn definitions. The resulting proof-specific definitions together with the CNF from the input instance can efficiently verify UNSAT using unmodified *drat-trim* [37].

4 Instantiation-Based Horn Approximation

This section describes how we derive a set of clauses sufficient to make theory lemmas DRAT-checkable. Section 4.1 introduces one-sided propositional definitions and motivates the goal of a compact, Horn-clause-based definition. Section 4.2 gives a translation from an arbitrary propositional definition of a monotonic predicate to a *monotonic definition*, as an intermediate step toward constructing the final proof-specific, Horn definition in Section 4.3.

4.1 One-Sided Propositional Definitions and Horn Clauses

Definition 2 (Propositional Definition). Let \mathbf{p} be the positive predicate atom of predicate p over Boolean arguments A . A propositional definition of \mathbf{p} , denoted as $\Sigma_{\mathbf{p}}$, is a CNF formula over variables $V \supseteq (\text{var}(\mathbf{p}) \cup A)$ such that for every truth assignment \mathcal{M} to the variables in A , (1) $\Sigma_{\mathbf{p}}|_{\mathcal{M}}$ is satisfiable and

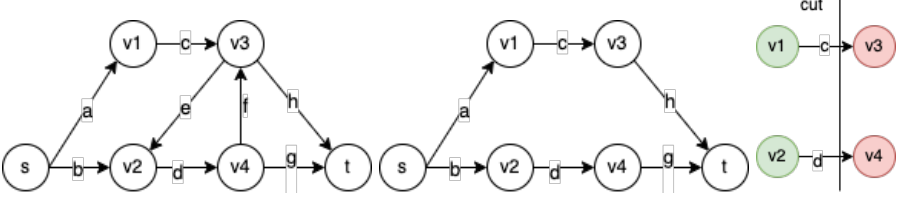


Fig. 2. Directed Graph for Running Example in Sec. 4. In the symbolic graph (left), the reachability predicate $reach_s^t$ is a function of the edge inputs a, \dots, h .

(2) $\Sigma_{\mathbf{p}} \models (\mathcal{M} \Rightarrow \mathbf{p})$ if and only if $p(\mathcal{M})$ is \top . The propositional definition of $\bar{\mathbf{p}}$ is defined analogously.

For example, the Tseitin-encoding of a logic circuit that computes $p(\mathcal{M})$ satisfies this definition. However, note that a propositional definition for \mathbf{p} can be one-sided: it is not required that $\Sigma_{\mathbf{p}} \models (\mathcal{M} \Rightarrow \bar{\mathbf{p}})$ when $p(\mathcal{M})$ is \perp . That case is handled by a separate propositional definition for $\bar{\mathbf{p}}$. We will see that this one-sidedness gives some freedom to admit more compact definitions.

Given a propositional definition $\Sigma_{\mathbf{p}}$, any theory lemma $\mathcal{M}_A \Rightarrow \mathbf{p}$ is a logical consequence of $\Sigma_{\mathbf{p}}$, but this might not be RUP checkable. One could prove $\Sigma_{\mathbf{p}} \models (\mathcal{M}_A \Rightarrow \mathbf{p})$ by calling a proof-generating SAT solver on $\Sigma_{\mathbf{p}} \wedge \mathcal{M}_A \Rightarrow \mathbf{p}$, i.e., bit-blasting the specific lemma, but we will see experimentally (in Sec. 6) that this works poorly. However, if the propositional definition is limited to Horn theory (i.e., each clause has at most one positive literal), then every SMT theory lemma *can* be proven by unit propagation:

Theorem 1. Let p be a positive monotonic predicate over input A , and let $\Sigma_{\mathbf{p}}^h$ be a propositional definition for the positive atom \mathbf{p} . If $\Sigma_{\mathbf{p}}^h$ is set of Horn clauses, then for any theory lemma $\mathcal{M}_A \Rightarrow \mathbf{p}$ where \mathcal{M}_A is a set of positive atoms from A , $\Sigma_{\mathbf{p}}^h \models (\mathcal{M}_A \Rightarrow \mathbf{p})$ if and only if $\Sigma_{\mathbf{p}}^h \vdash_1 (\mathcal{M}_A \Rightarrow \mathbf{p})$.

Proof. Suppose $\Sigma_{\mathbf{p}}^h \models (\mathcal{M}_A \Rightarrow \mathbf{p})$, then $\Sigma_{\mathbf{p}}^h \wedge (\mathcal{M}_A \wedge \bar{\mathbf{p}})$ is unsatisfiable. Since $\mathcal{M}_A \wedge \bar{\mathbf{p}}$ is equivalent to a set of unit clauses, $\Sigma_{\mathbf{p}}^h \wedge (\mathcal{M}_A \wedge \bar{\mathbf{p}})$ still contains only Horn clauses, so satisfiability can be determined by unit propagation. \square

Example 1. Let $reach_s^t$ be the reachability predicate for the directed graph shown in Fig. 2 (left). The definition schema for graph reachability in Sec. 5 yields the following set of Horn clauses: $\Sigma_{reach_s^t}^h := (1) \bar{s} \vee \bar{a} \vee v1, (2) \bar{v1} \vee \bar{c} \vee v3, (3) \bar{v3} \vee \bar{h} \vee t, (4) \bar{s} \vee \bar{b} \vee v2, (5) \bar{v3} \vee \bar{e} \vee v2, (6) \bar{v2} \vee \bar{d} \vee v4, (7) \bar{v4} \vee \bar{f} \vee v3, (8) \bar{v4} \vee \bar{g} \vee t, (9) \bar{t} \vee reach_s^t, (10) s$, where $v1, \dots, v5, s$, and t are auxiliary variables. Any theory lemma of the form $\mathcal{M}_A \Rightarrow \mathbf{p}$, e.g., $\bar{a} \vee \bar{c} \vee \bar{h} \vee reach_s^t$, can be proven from $\Sigma_{reach_s^t}^h$ via unit propagation. Also, note that one-sidedness allows a simpler definition, despite the cycle in the graph, e.g., consider assignment $\mathcal{M} = \{\bar{a}, \bar{b}, \bar{c}, \bar{d}, e, f, g, h\}$. Then, $reach_s^t = \perp$, but $\Sigma_{reach_s^t}^h \not\models (\mathcal{M} \Rightarrow reach_s^t)$.

Horn theory has limited expressiveness, but it is always sufficient to encode a propositional definition for any SMT theory: Given a monotonic predicate

atom \mathbf{p} , we can always encode a *Horn propositional definition* $\Sigma_{\mathbf{p}}^h$ as the conjunction of all valid theory lemmas from the theory of \mathbf{p} . This is because every theory lemma is restricted to the form $(\mathcal{M}_A \Rightarrow \mathbf{p})$, where \mathcal{M}_A is a set of positive atoms (due to monotonicity). Hence, $\Sigma_{\mathbf{p}}^h$ is a set of Horn clauses. However, such a naïve encoding blows up exponentially. Instead, we will seek a compact Horn definition $\Sigma_{\mathbf{p}}^h$ that *approximates* a non-Horn propositional definition $\Sigma_{\mathbf{p}}$:

Definition 3 (Horn Upper-Bound). *Let $\Sigma_{\mathbf{p}}$ be a propositional definition of \mathbf{p} . A set of Horn clauses $\Sigma_{\mathbf{p}}^{h\uparrow}$ is a Horn upper-bound if $\Sigma_{\mathbf{p}} \models \Sigma_{\mathbf{p}}^{h\uparrow}$.*

For the strongest proving power, we want the tightest Horn upper-bound possible. Unfortunately, the least Horn upper-bound of a non-Horn theory can still contain exponentially many Horn clauses [35]. Fortunately, we don't actually need a Horn upper-bound on the *exact* theory definition, but only of enough of the definition to prove the fixed set of theory lemmas that constitute the proof obligations. This motivates the next definition.

Definition 4 (Proof-Specific Horn Definition). *Given an exact definition $\Sigma_{\mathbf{p}}$ and a set of theory lemmas $\mathbb{O} := \{C_1, \dots, C_n\}$ from the theory of \mathbf{p} , a proof-specific Horn definition of \mathbf{p} is a Horn upper-bound $\Sigma_{\mathbf{p}}^{h\uparrow}$ of $\Sigma_{\mathbf{p}}$ such that $\Sigma_{\mathbf{p}}^{h\uparrow} \vdash_1 C$ for every $C \in \mathbb{O}$.*

Our goal in the next two subsections is how to derive such compact, proof-specific Horn definitions.

Example 2. Continuing Ex. 1, given a proof obligation \mathbb{O} with two theory lemmas: $\{\bar{\mathbf{a}} \vee \bar{\mathbf{c}} \vee \bar{\mathbf{h}} \vee \mathbf{reach}_{\mathbf{s}}^{\mathbf{t}}, \bar{\mathbf{b}} \vee \bar{\mathbf{d}} \vee \bar{\mathbf{g}} \vee \mathbf{reach}_{\mathbf{s}}^{\mathbf{t}}\}$, the subset of Horn clauses with IDs (1), (2), (3), (4), (6), (8), (9) and (10) is a proof-specific Horn definition for $\mathbf{reach}_{\mathbf{s}}^{\mathbf{t}}$, which can be visualized in Fig. 2 (middle).

Given a proof obligation \mathbb{O} , we can make all theory lemmas in \mathbb{O} DRAT checkable if we have exact propositional definitions for the theories and if we can dynamically transform them into compact, proof-specific Horn definitions at the time of proof checking. We simply add these additional clauses to the input of the DRAT-proof-checker.

4.2 Monotonic Definitions

The derivation of compact, proof-specific Horn definitions from arbitrary propositional definitions is a two-step process: we first show that every propositional definition for a monotonic predicate atom can be converted into a monotonic definition of linear size (this section), and then use theory lemmas in the proof obligations to create the Horn approximation of the definition (Sec. 4.3).

Definition 5 (Monotonic Definition). *Let a monotonic predicate p over input A be given. A CNF formula $\Sigma_{\mathbf{p}}^+$ is a monotonic definition of the positive predicate atom \mathbf{p} if $\Sigma_{\mathbf{p}}^+$ is a propositional definition of \mathbf{p} , and it satisfies the following syntax restrictions: (1) $\Sigma_{\mathbf{p}}^+$ does not contain positive atoms from A , (2) $\Sigma_{\mathbf{p}}^+$ does not contain $\bar{\mathbf{p}}$, and (3) \mathbf{p} appears only in Horn clauses. The monotonic definition for $\bar{\mathbf{p}}$ is defined analogously.*

We now define the procedure, **MONOT**, for transforming a propositional definition into a linear-size monotonic definition:

Definition 6 (Monotonic Transformation). *Let a monotonic predicate p over input A and a propositional definition $\Sigma_{\mathbf{p}}$ for the positive predicate atom \mathbf{p} be given. $\text{MONOT}(\mathbf{p}, \Sigma_{\mathbf{p}})$ is the result of the following transformations on $\Sigma_{\mathbf{p}}$: (1) replace every occurrence of an input atom (\mathbf{a} for $a \in A$) in $\Sigma_{\mathbf{p}}$ with a new atom \mathbf{a}' ($\bar{\mathbf{a}}$ is replaced with $\overline{\mathbf{a}'}$), (2) replace every occurrence of \mathbf{p} and $\bar{\mathbf{p}}$ with \mathbf{p}' and $\overline{\mathbf{p}'}$ respectively, and (3) add the following Horn clauses: $\mathbf{a} \Rightarrow \mathbf{a}'$ for every $a \in A$, and $\mathbf{p}' \Rightarrow \mathbf{p}$.*

Theorem 2 (Correctness of Monotonic Transformation). *Given a monotonic predicate p over input A and the monotonic predicate atom \mathbf{p} , if we have any propositional definition $\Sigma_{\mathbf{p}}$ with n clauses, then $\text{MONOT}(\mathbf{p}, \Sigma_{\mathbf{p}})$ results in a monotonic definition $\Sigma_{\mathbf{p}}^+$ with at most $n + |A| + 1$ clauses.*

The proof of Theorem 2 is in the extended version of this paper. The correctness relies on the fact that the predicate p is indeed monotonic, and that our propositional definitions need only be one-sided. If the monotonic definition is already in Horn theory, it can be used directly verify theory lemmas via RUP; otherwise, we proceed to Horn approximation, described next.

4.3 Instantiation-Based, Proof-Specific Horn Definition

We present the transformation from monotonic definitions into proof-specific Horn definitions. The transformation exploits the duality between predicates' positive and negative definitions.

Lemma 1 (Duality). *Let p be a monotonic predicate over Boolean arguments A . Suppose $\Sigma_{\mathbf{p}}$ and $\Sigma_{\bar{\mathbf{p}}}$ are positive and negative propositional definitions, respectively. For every assignment \mathcal{M} to the variables in A :*

1. $\Sigma_{\mathbf{p}} \models (\mathcal{M} \Rightarrow \mathbf{p})$ if and only if $\Sigma_{\bar{\mathbf{p}}} \wedge \mathcal{M} \wedge \mathbf{p}$ is satisfiable.
2. $\Sigma_{\bar{\mathbf{p}}} \models (\mathcal{M} \Rightarrow \bar{\mathbf{p}})$ if and only if $\Sigma_{\mathbf{p}} \wedge \mathcal{M} \wedge \bar{\mathbf{p}}$ is satisfiable.

The proof of Lemma 1 is in the extended version of this paper. The duality of the positive ($\Sigma_{\mathbf{p}}$) and negative ($\Sigma_{\bar{\mathbf{p}}}$) definitions allows us to over-approximate positive (negative) definitions by instantiating the negative (positive) definitions.

Example 3. Returning to Ex. 1 and Fig. 2, consider the assignment $M = \{\mathbf{a}, \mathbf{b}, \bar{\mathbf{c}}, \bar{\mathbf{d}}, \mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$. Since s cannot reach t under this assignment, any propositional definition $\Sigma_{\text{reach}_s^t}$ must imply $M \Rightarrow \text{reach}_s^t$. Dually, $\Sigma_{\text{reach}_s^t}^h \wedge M \wedge \text{reach}_s^t$ is satisfiable, e.g., $\{\mathbf{s}, \mathbf{v1}, \mathbf{v2}, \bar{\mathbf{v3}}, \bar{\mathbf{v4}}, \bar{\mathbf{t}}\}$.

Lemma 2 (Instantiation-Based Upper-Bound). *Let a predicate p over input A and a positive definition $\Sigma_{\mathbf{p}}$ be given. For any partial assignment M' over $\text{var}(\Sigma_{\mathbf{p}}) \setminus (\text{var}(p) \cup A)$, $\Sigma_{\mathbf{p}}|_{M' \cup \bar{\mathbf{p}}} \Rightarrow \bar{\mathbf{p}}$ is an over-approximation of $\Sigma_{\bar{\mathbf{p}}}$.⁶*

⁶ Note that $\Sigma_{\mathbf{p}}|_{M'}$ is encoded in CNF, so to compactly (i.e., linear-size) encode $\Sigma_{\mathbf{p}}|_{M'} \Rightarrow \bar{\mathbf{p}}$ in CNF, we introduce a new literal l_i for each clause $C_i \in \Sigma_{\mathbf{p}}|_{M'}$, create clauses $\bar{c}_{ij} \vee l_i$ for each literal $c_{ij} \in C_i$, and add clause $\bar{l}_1 \vee \bar{l}_2 \vee \dots \vee \bar{l}_n \vee \bar{\mathbf{p}}$.

The proof of Lemma 2 (in the extended paper) relies on the duality in Lemma 1. Lemma 2 enables upper-bound construction and paves the way for constructing an instantiation-based Horn upper-bound of a monotonic definition.

Lemma 3 (Instantiation-Based Horn Upper-Bound). *Given a monotonic predicate p over input A and a positive monotonic definition $\Sigma_{\mathbf{p}}^+$, let X represent the set of auxiliary variables: $\text{var}(\Sigma_{\mathbf{p}}^+) \setminus (A \cup \text{var}(p))$. For any complete satisfying assignment $M_{X \cup A}$ to $\Sigma_{\mathbf{p}}^+|_{\bar{\mathbf{p}}}$, the formula $(\Sigma_{\mathbf{p}}^+|_{\bar{\mathbf{p}} \cup \mathcal{M}_X}) \Rightarrow \bar{\mathbf{p}}$ serves as a Horn upper-bound for any propositional definition of $\bar{\mathbf{p}}$, where \mathcal{M}_X is a partial assignment derived from $M_{X \cup A}$ for the auxiliary variables X .*

(Proof in the extended paper.) Note that the instantiation-based Horn upper-bound of a negative predicate atom $\bar{\mathbf{p}}$ is constructed from a monotonic definition of the positive predicate atom $\Sigma_{\mathbf{p}}^+$, and vice-versa.

For a given theory lemma, the instantiation-based Horn upper-bound construction (Lemma 3) enables the verification of the theory lemma if we can find a sufficient “witness” \mathcal{M}_X for the instantiation. We now prove that a witness always exists for every valid theory lemma and does not exist otherwise.

Theorem 3 (Lemma-Specific Horn Upper-Bound). *Let a monotonic predicate p over input A , a monotonic definition $\Sigma_{\mathbf{p}}^+$ and a lemma in the form $\mathcal{M}_A \Rightarrow \bar{\mathbf{p}}$ be given. We denote X as the set of auxiliary variables: $\text{var}(\Sigma_{\mathbf{p}}^+) \setminus (A \cup \text{var}(p))$. The lemma $\mathcal{M}_A \Rightarrow \bar{\mathbf{p}}$ is in the theory of $\bar{\mathbf{p}}$ if and only if there exists an assignment \mathcal{M}_X on X such that: (1) $\Sigma_{\mathbf{p}}^+|_{\bar{\mathbf{p}} \cup \mathcal{M}_X \cup \mathcal{M}_A}$ is satisfiable and (2) $(\Sigma_{\mathbf{p}}^+|_{\bar{\mathbf{p}} \cup \mathcal{M}_X} \Rightarrow \bar{\mathbf{p}}) \vdash_1 (\mathcal{M}_A \Rightarrow \bar{\mathbf{p}})$.*

(Proof in the extended paper.) Theorem 3 states that a lemma-specific Horn upper-bound for a theory lemma $\mathcal{M}_A \Rightarrow \bar{\mathbf{p}}$ can be constructed by instantiating the monotonic definition using a “witness” assignment \mathcal{M}_X .⁷ The witness could be obtained by performing SAT solving on the formula $\Sigma_{\mathbf{p}}^+|_{\mathcal{M}_A^+ \cup \bar{\mathbf{p}}}$ (where \mathcal{M}_A^+ is the extension of \mathcal{M}_A by assigning unassigned input variables in A to \top (Sec. 2.2)). However, in practice, a better approach is to modify the SMMT solver to produce the witness during the derivation of theory lemmas. In Section 5, we provide examples of witnesses for commonly used monotonic predicates.

Note that the witness is not part of the trusted foundation for the proof. An incorrect witness might not support verification of a theory lemma, but if a theory lemma is verified using a specific witness \mathcal{M}_X , Theorem 3 guarantees that the lemma is valid.

Example 4. Continuing the example, let a theory lemma $L := \mathbf{c} \vee \mathbf{d} \vee \overline{\text{reach}}_{\mathbf{g}}^{\mathbf{t}}$ be given. To derive a lemma-specific Horn upper-bound for $\Sigma_{\overline{\text{reach}}_{\mathbf{g}}^{\mathbf{t}}}^+$, we first obtain a witness \mathcal{M}_X by finding a satisfying assignment to the formula $\Sigma_{\overline{\text{reach}}_{\mathbf{g}}^{\mathbf{t}}}^h \wedge M \wedge \overline{\text{reach}}_{\mathbf{g}}^{\mathbf{t}}$, where $M := \{\mathbf{a}, \mathbf{b}, \bar{\mathbf{c}}, \bar{\mathbf{d}}, \mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$ (by assigning the unassigned

⁷ Instead of instantiating a complete assignment on every auxiliary variable in X , a partial instantiation is sufficient so long as it determines the assignments on the other variables.

input variables in L to \top). Since M is a complete assignment to the edge variables, the graph is fully specified, and a suitable witness \mathcal{M}_X can be efficiently computed using a standard graph-reachability algorithm, to compute the reachability status of each vertex. The witness \mathcal{M}_X is $\{\mathbf{s}, \mathbf{v1}, \mathbf{v2}, \overline{\mathbf{v3}}, \mathbf{v4}, \overline{\mathbf{t}}\}$. Following the construction in Theorem 3, the formula $\Sigma_{\text{reach}_s^t}^h | \overline{\text{reach}_s^t} \cup \mathcal{M}_X$ simplifies to two (unit) clauses: $\overline{\mathbf{c}}$ and $\overline{\mathbf{d}}$ (from clauses (2) and (6) in Ex. 1), which can be visualized as the cut in Fig. 2 (right). The lemma-specific Horn upper bound $\Sigma_{\text{reach}_s^t}^h | \overline{\text{reach}_s^t} \cup \mathcal{M}_X \Rightarrow \text{reach}_s^t$ is, therefore, $\overline{\mathbf{c}} \wedge \overline{\mathbf{d}} \Rightarrow \text{reach}_s^t$, which in this example is already CNF, but more generally, we would introduce two literals to encode the implication: $\{\mathbf{c} \vee \overline{\mathbf{l}_1}, \mathbf{d} \vee \overline{\mathbf{l}_2}, \mathbf{l}_1 \vee \mathbf{l}_2 \vee \overline{\text{reach}_s^t}\}$. The lemma-specific Horn upper-bound is dual-Horn and implies the theory lemma L by unit propagation.

From the lemma-specific Horn upper-bounds, we construct the proof-specific Horn definition by combining the lemma-specific Horn upper-bounds for all lemmas in the proof obligations.

In summary, to efficiently verify SMMT theory lemmas, we propose the following approach: (1) define the propositional definitions (in CNF) for the atoms of theory predicates; (2) transform the definitions into monotonic definitions offline; (3) during proof checking, approximate a proof-specific Horn definition (if not already Horn) from the constructed monotonic definition using theory lemmas in the proof; (4) combine the proof-specific definition together and verify the proof via RUP. The only theory-specific, trusted foundation for the proof is the definition for the theory atoms. (The extended version of this paper contains a figure to help visualize this workflow.)

Example 5. Summarizing, the positive propositional definition $\Sigma_{\text{reach}_s^t}$ in Ex. 1 is already Horn, so is sufficient for verifying via DRAT any SMMT lemmas that imply reach_s^t . To verify lemmas that imply $\overline{\text{reach}_s^t}$, we can compute a proof-specific definition of $\overline{\text{reach}_s^t}$ from $\Sigma_{\text{reach}_s^t}$ using Theorem 3.

Remark 1. The only trusted basis of our approach are the propositional definitions of theory atoms. For the monotonic theories in the section 5, we considered the definitions intuitively understandable, and therefore sufficiently trustworthy. But to further increase confidence, propositional definitions can be validated using techniques from hardware validation/verification, e.g., simulation to sanity-check general behavior, equivalence checking against known-good circuits, etc.

5 Example Propositional Definitions

In this section, we illustrate the monotonic definitions for the most commonly used monotonic predicates. Due to space constraints, we present only graph reachability here in detail, and only sketch bit-vector comparison and summation, and max-flow. Full definitions for those theories are in the extended version of this paper.

Graph Reachability: Given a graph $G = (V, E)$ where V and E are sets of vertices and edges, respectively, as discussed in Sect. 2, the graph reachability theory contains the reachability predicate reach_u^v for $u, v \in V$ over input

$e_1, e_2 \dots e_n \in E$. For convenience, we refer to the positive edge atom for the edge from vertex i to vertex j as $\mathbf{e}_{i \rightarrow j}$. The predicate is positively monotonic for E , and the monotonic definition for the positive predicate atom \mathbf{reach}_u^v contains the clauses:

1. $\overline{reach^i} \vee \overline{\mathbf{e}_{i \rightarrow j}} \vee reach^j$ for every edge $e_i^j \in E$ and the unit clause $reach^u$
2. $\overline{reach^v} \vee \mathbf{reach}_u^v$

The monotonic definition introduces a reachability atom $reach^i$ for every $i \in V$ and asserts the fact that u is reachable from itself. For every edge (i, j) , if the edge (i, j) is enabled ($\mathbf{e}_{i \rightarrow j}$) and i is reachable ($reach^i$), then j must also be reachable ($reach^j$). The predicate atom \mathbf{reach}_u^v is implied by the reachability of v ($reach^v$). The definition is monotonic since it only contains negative edge atoms. Moreover, the definition is already a Horn definition and can be used directly for proving theory lemmas in the theory of \mathbf{reach}_u^v without the need for transformation into a proof-specific Horn definition. The size of the definition is $O(|E|)$.

Instead of defining the monotonic definition for the negative predicate atom \mathbf{reach}_u^v , we construct its proof-specific definition from the monotonic definition of the positive predicate atom \mathbf{reach}_u^v . For each theory lemma in the proof, the witness for constructing the lemma-specific Horn upper-bound is the reachability status ($reach^i$) of every vertex $i \in V$, which is efficiently computed in the SMMT solver using standard graph-reachability algorithms.

Bit-Vector Comparison (sketch): The positive definition is just the Tseitin encoding of a typical bit-vector comparison circuit, with some simplification due to being one-sided: For each bit position i , we introduce auxiliary variables ge_i and gt_i , which indicate that the more-significant bits from this position have already determined vector \vec{a} to be \geq or $>$ \vec{b} , respectively. Simple clauses compute ge_{i-1} and gt_{i-1} from ge_i and gt_i and the bits at position $i - 1$ of \vec{a} and \vec{b} . The negative definition is similar. These are both Horn, so can be used without further transformation into proof-specific Horn definitions.

Bit-Vector Summation and Comparison (sketch): These are basically Tseitin encodings of ripple-carry adders, combined with the comparison theory above — using Def. 6 to handle the fact that the the Tseitin encodings of the XOR gates in the adders are non-monotonic with respect to the input bit-vectors. The resulting propositional definitions are not Horn, so we use witnesses to construct lemma-specific Horn definitions. The witnesses come from the SMMT solver maintaining lower and upper bounds on the possible values of the bit-vectors, e.g., a witness for $\sum \vec{A} \geq \sum \vec{B}$ are lower bounds for the vectors in \vec{A} and upper bounds for the vectors in \vec{B} such that their sums make the inequality true. (*Mutadis mutandis* for the negative witness.)

Max-Flow (sketch): For the positive definition (that the max-flow exceeds some value), we introduce auxiliary bit-vectors to capture the flow assigned to each edge. We use the bit-vector theories to ensure that the flows do not exceed the edge capacities, that each node's (except the source) outgoing flows do not exceed the incoming flows (equality is unnecessary due to the one-sidedness), and

that the flow to the sink exceeds the target value. For the negative definition, we exploit the famous max-flow/min-cut duality. We introduce an auxiliary variable $incut_e$ for each edge. We use the graph reachability theory to ensure that the edges in the cut separate the source from the sink, and the bit-vector summation theory to ensure that the capacity of the cut does not exceed the target max-flow value. Both the positive and negative definitions are not Horn, so require instantiation-based upper-bounds. The witnesses are the flow values or the cuts, and are easily computed by the SMMT solver.

6 Experimental Evaluation

To evaluate our proposed method, we implemented it as shown earlier in Fig. 1 (Sec. 3). We call our implementation *MonoProof* (available at <https://github.com/NickF0211/MonoProof>).

The two basic questions of any proof-generating SAT/SMT solver are: (1) how much overhead does the support for proofs add to the solving time, and (2) how efficiently can a proof be prepared from the proof log, and verified? For the first question, we compare the runtime of unmodified MonoSAT versus the MonoSAT that we have extended to produce proof certificates. For the second question, we need a baseline of comparison. MonoProof is the first proof-generating SMMT solver, so there is no obvious comparison. However, since SMMT theories are finite-domain, and bit-blasting (i.e., adding clauses that encode the theory predicates to the problem instance and solving via a propositional SAT solver) is a standard technique for finite-domain theories, we compare against bit-blasting. Arguably, this comparison is unfair, since MonoSAT outperforms bit-blasting when *solving* SMMT theories [9]. Thus, as an additional baseline, we propose an obvious hybrid of SMMT and bit-blasting, which we dub Lemma-Specific Bit-Blasting (LSBB): we run MonoProof until the core theory lemmas have been extracted, benefitting from MonoSAT’s fast solving time, but then instead of using our techniques from Sec. 4, we bit-blast only the core theory lemmas.⁸

We ran experiments on 3GHZ AMD Epyc 7302 CPUs with 512GB of DDR4 RAM, with a timeout of 1 hour and memory limit of 64GB. For the bit-blasting SAT solver, we use the state-of-the-art SAT solver Kissat [13]. In all cases, the proof is verified with standard DRAT-trim [37].

6.1 Benchmarks

We wish to evaluate scalability on real, industrial problems arising in practice. MonoProof has successfully generated and verified industrial UNSAT proofs for

⁸ We implemented this both via separate SAT calls per lemma; and also by providing all lemmas in a single SAT call (with auxiliary variables to encode the resulting DNF), to allow the solver to re-use learned clauses on different lemmas. The latter approach generally worked better, so we report those results, but (spoiler) neither worked well.

a set of hard, unsatisfiable Tiros [2,8] queries collected in production use at AWS over a multi-week period. However, these instances are proprietary and cannot be published, making them irreproducible by others. Instead, we evaluate on two sets of benchmarks that we can publicly release (also at <https://github.com/NickF0211/MonoProof>):

Network Reachability Benchmarks. These are synthetic benchmarks that mimic the real-world problems solved by Tiros, without disclosing any proprietary information. Network reachability is the problem of determining whether a given pair of network resources (source and destination) can communicate. The problem is challenging because network components can intercept, transform, and optionally re-transmit packets traveling through the network (e.g., a firewall or a NAT gateway). Network components come in various types, each with their own complex behaviors and user-configurable network controls. In these benchmarks, we abstract to two types of intermediate components: simple and transforming. Simple components relay an incoming packet as long as its destination address belongs to a certain domain, expressed in terms of a network CIDR (Classless Interdomain Routing), e.g., 10.0.0.0/24. Transforming network components intercept an incoming packet and rewrite the source address and ports to match their own before re-transmitting it. The simple network components are akin to subnets, VPCs, and peering connections; transforming network components are a highly abstracted version of load balancers, NAT gateways, firewalls, etc. The SMT encoding uses the theories of bit vectors and of graph reachability. The network packets are symbolically represented using bit vectors, and the network is modeled as a symbolic graph. Network behavior is modeled as logical relations between packets and elements in the network graph. Unsatisfiability of a query corresponds to unreachability in the network: for all possible packet headers that the source could generate, and for all possible paths connecting the source to the destination, the combined effect of packet transformations and network controls placed along the path cause the packet to be dropped from the network before it reaches its destination.

We generated 24 instances in total, varying the size and structure of the randomly generated network. Graph sizes ranged from 1513 to 15524 (average 5485) symbolic edges.

Escape Routing Benchmarks. Escape routing is the problem of routing all the signals from a component with extremely densely packed I/O connections (e.g., the solder bumps on a Ball-Grid Array (BGA)) to the periphery of the component, where other routing techniques can be used. For a single-layer printed circuit board (PCB), escape routing is optimally solvable via max-flow, but real chips typically require multiple layers. The multi-layer problem is difficult because the vias (connections between layers) are wider than the wires on a layer, disrupting what routes are possible on that layer. Bayless et al. [11] proposed a state-of-the-art solution using SMMT: max-flow predicates determine routability for each layer on symbolic graphs, whose edges are enabled/disabled by logical constraints capturing the design rules for vias.

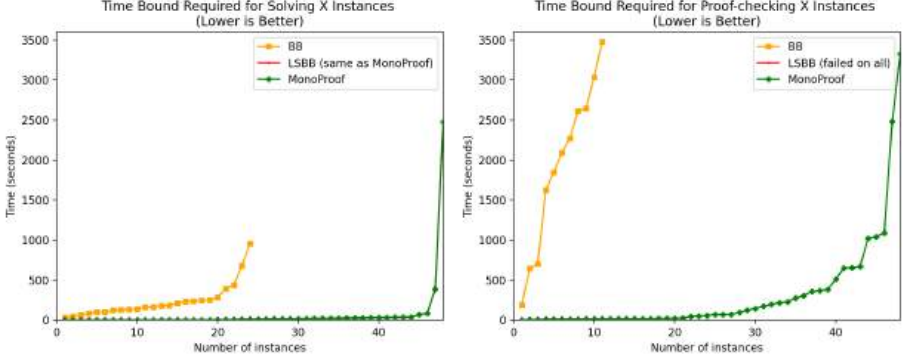


Fig. 3. Cactus Plots for Solving (left) and Proof Preparation&Checking (right). Each point is the runtime for one instance, so the plot shows the number of instances (x -axis) that ran in less than any time bound (y -axis). BB denotes standard bit-blasting; LSBB, lemma-specific bit-blasting; and MonoProof is our new method. The left graph shows that MonoProof (and LSBB, which uses MonoProof’s solver) is vastly faster than bit-blasting for solving the instances. The right graph shows that MonoProof is also vastly faster than bit-blasting for proving the result; LSBB timed-out on all proofs.

In [11], 24 commercial BGAs were analyzed under two different via technologies and different numbers of layers. For our benchmark set, we select all configurations where the *provable* minimum number of layers were reported. This results in 24 unsatisfiable SMMT problems instances (routing with one fewer layer than the minimum), which exercise the bit-vector and max-flow theories. Graph sizes ranged from 193994 to 3084986 (average 717705) symbolic edges.

6.2 Results

Returning to the two questions for our evaluation:

1. *The solver overhead of our proof certificate generation is minimal.* On the network reachability benchmarks, the geometric mean (GM) runtime overhead was 14.10% (worst case 28.8%). On the escape routing benchmarks, the GM runtime overhead was only 1.11% (the worst case 5.71%). (The lower overhead is because MonoSAT spent more time learning theory lemmas vs. recording them in the proof.) The overall GM runtime overhead across all benchmarks was 7.41%. These overhead figures are comparable to state-of-the-art, proof-generating SAT solvers, which is not surprising, since our proof certificates are essentially the same as a DRAT proof certificate in SAT. This compares favorably with the solver overhead of heavier-weight, richer, and more expressive SMT proof certificates like LFSC [34].

2. *MonoProof’s time to prepare and check a proof of unsatisfiability is markedly faster than standard bit-blasting or lemma-specific bit-blasting.* Fig. 3 summarizes our results. (A full table is in the extended version of this paper.) The left

graph shows solving times (with proof logging). Since the proof-logging overhead is so low for both bit-blasting (Kissat generating DRAT) and MonoProof, these results are consistent with prior work showing the superiority of the SMMT approach for *solving* [9]. Note that bit-blasting (BB) solved all 24 network reachability instances, but failed to solve any of the 24 escape routing instances in the 1hr timeout. Lemma-specific bit-blasting (LSBB) and MonoProof share the same solving and proof-logging steps. The right graph shows proof-checking times (including BackwardCheck and proof-specific Horn upper-bound construction for MonoProof). Here, BB could proof-check only 11/24 reachability instances that it had solved. Restricting to only the 11 instances that BB proof-checked, MonoProof was at least $3.7\times$ and geometric mean (GM) $10.2\times$ faster. LSBB timed out on all 48 instances. Summarizing, MonoProof solved and proved all 48 instances, whereas BB managed only 11 instances, and LSBB failed to prove any.

The above results were with our modified BackwardCheck enabled (*drat-trim-theory* in Fig. 1). Interestingly, with BackwardCheck disabled, MonoProof ran even faster on 37/48 benchmarks (min speedup $1.03\times$, max $6.6\times$, GM $1.7\times$). However, enabling BackwardCheck ran faster in 10/48 cases (min speedup $1.02\times$, max $7.9\times$, GM $1.6\times$), and proof-checked one additional instance (69 sec. vs. 1hr timeout). The modified BackwardCheck is a useful option to have available.

7 Conclusion

We have introduced the first efficient proof-generating method for SMMT. Our approach uses propositional definitions of the theory semantics and derives compact, proof-specific Horn-approximations sufficient to verify the theory lemmas via RUP. The resulting pure DRAT proofs are checkable via well-established (and even machine verified) tools. We give definitions for the most common SMMT theories, and experimental results on industrial-scale problems demonstrate that the solving overhead is minimal, and the proof preparation and checking times are vastly faster than the alternative of bit-blasting.

The immediate line of future work is to support additional finite domain monotonic theories, such as richer properties on pseudo-boolean reasoning. We also aim to apply our approach to support monotonic theories beyond finite domains. In addition, we plan to extend our proof support to emerging proof format such as LRAT [18] and FRAT [3] that enable faster proof checking.

Acknowledgments

Nick Feng was supported in part by an Amazon Research Award. Alan Hu was supported in part by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would like to thank Marijn Heule for insightful feedback and mentorship during Nick’s internship at Amazon Web Services in 2021. We also thank Dan Dacosta, Nadia Labai, and Nate Launchbury for reviewing earlier drafts of this work.

References

1. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, J., Shao, Z. (eds.) *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings. Lecture Notes in Computer Science*, vol. 7086, pp. 135–150. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_12, https://doi.org/10.1007/978-3-642-25379-9_12
2. Backes, J., Bayless, S., Cook, B., Dodge, C., Gacek, A., Hu, A.J., Kahsai, T., Kocik, B., Kotelnikov, E., Kukovec, J., McLaughlin, S., Reed, J., Rungta, N., Sizemore, J., Stalzer, M., Srinivasan, P., Subotić, P., Varming, C., Whaley, B.: Reachability analysis for AWS-based networks. In: Dillig, I., Tasiran, S. (eds.) *International Conference on Computer Aided Verification (CAV)*. pp. 231–241. Springer (2019)
3. Baek, S., Carneiro, M., Heule, M.J.H.: A Flexible Proof Format for SAT Solver-Elaborator Communication. In: Groote, J.F., Larsen, K.G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems — 27th International Conference, TACAS 2021. Lecture Notes in Computer Science*, vol. 12651, pp. 59–75. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_4, https://doi.org/10.1007/978-3-030-72016-2_4
4. Barbosa, H., Blanchette, J., Fleury, M., Fontaine, P., Schurr, H.J.: Better SMT proofs for easier reconstruction. In: *AITP 2019-4th Conference on Artificial Intelligence and Theorem Proving* (2019)
5. Barbosa, H., Blanchette, J.C., Fontaine, P.: Scalable Fine-Grained Proofs for Formula Processing. In: de Moura, L. (ed.) *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science*, vol. 10395, pp. 398–412. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_25, https://doi.org/10.1007/978-3-319-63046-5_25
6. Barrett, C., De Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. *All about proofs, Proofs for all* **55**(1), 23–44 (2015)
7. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science*, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14, https://doi.org/10.1007/978-3-642-22110-1_14
8. Bayless, S., Backes, J., DaCosta, D., Jones, B., Launchbury, N., Trentin, P., Jewell, K., Joshi, S., Zeng, M., Mathews, N.: Debugging Network Reachability with Blocked Paths. In: *International Conference on Computer Aided Verification (CAV)*. pp. 851–862. Springer (2021)
9. Bayless, S., Bayless, N., Hoos, H., Hu, A.: SAT modulo monotonic theories. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. vol. 29 (2015)
10. Bayless, S., Bayless, N., Hoos, H.H., Hu, A.J.: SAT Modulo Monotonic Theories. In: Bonet, B., Koenig, S. (eds.) *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, January 25-30, 2015, Austin, Texas, USA. pp. 3702–3709. AAAI Press (2015), <http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9951>
11. Bayless, S., Hoos, H.H., Hu, A.J.: Scalable, high-quality, SAT-based multi-layer escape routing. In: Liu, F. (ed.) *Proceedings of the 35th International Conference*

- on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016. p. 22. ACM (2016). <https://doi.org/10.1145/2966986.2967072>, <https://doi.org/10.1145/2966986.2967072>
12. Bayless, S., Kodirov, N., Iqbal, S.M., Beschastnikh, I., Hoos, H.H., Hu, A.J.: Scalable Constraint-Based Virtual Data Center Allocation. *Artif. Intell.* **278**(C) (jan 2020). <https://doi.org/10.1016/j.artint.2019.103196>, <https://doi.org/10.1016/j.artint.2019.103196>
 13. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling, and Treengeling Entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Jarvisalo, M., Suda, M. (eds.) *SAT Competition 2020 — Solver and Benchmark Descriptions*. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
 14. Böhme, S.: Proof reconstruction for Z3 in Isabelle/HOL. In: 7th International Workshop on Satisfiability Modulo Theories (SMT’09) (2009)
 15. Böhme, S., Weber, T.: Fast LCF-Style Proof Reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6172, pp. 179–194. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_14, https://doi.org/10.1007/978-3-642-14052-5_14
 16. Bouton, T., Oliveira, D.C.B.D., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5663, pp. 151–156. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_12, https://doi.org/10.1007/978-3-642-02959-2_12
 17. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Donaldson, A.F., Parker, D. (eds.) *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*. Lecture Notes in Computer Science, vol. 7385, pp. 248–254. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19, https://doi.org/10.1007/978-3-642-31759-0_19
 18. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient Certified RAT Verification. In: de Moura, L. (ed.) *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*. Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_14, https://doi.org/10.1007/978-3-319-63046-5_14
 19. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient Certified Resolution Proof Checking. In: Legay, A., Margaria, T. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 118–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
 20. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. *J. ACM* **7**(3), 201–215 (1960). <https://doi.org/10.1145/321033.321034>, <http://doi.acm.org/10.1145/321033.321034>
 21. Fleury, M., Schurr, H.: Reconstructing veriT Proofs in Isabelle/HOL. In: Reis, G., Barbosa, H. (eds.) *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019. EPTCS*, vol. 301, pp. 36–50 (2019). <https://doi.org/10.4204/EPTCS.301.6>, <https://doi.org/10.4204/EPTCS.301.6>

22. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL (T): Fast decision procedures. In: International Conference on Computer Aided Verification. pp. 175–188. Springer (2004)
23. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reason.* **58**(1), 3–31 (2017). <https://doi.org/10.1007/s10817-016-9388-y>, <https://doi.org/10.1007/s10817-016-9388-y>
24. Goldberg, E., Novikov, Y.: Verification of Proofs of Unsatisfiability for CNF Formulas. In: Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1. p. 10886. DATE '03, IEEE Computer Society, USA (2003)
25. Gurfinkel, A., Vizel, Y.: DRUPing for interpolates. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21–24, 2014. pp. 99–106. IEEE (2014). <https://doi.org/10.1109/FMCAD.2014.6987601>, <https://doi.org/10.1109/FMCAD.2014.6987601>
26. Heule, M.J.H., Kiesl, B., Biere, A.: Strong Extension-Free Proof Systems. *J. Automated Reasoning* **64**, 533–554 (2020). <https://doi.org/10.1007/s10817-019-09516-0>
27. Heule, M.J., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: 2013 Formal Methods in Computer-Aided Design. pp. 181–188. IEEE (2013)
28. Klenze, T., Bayless, S., Hu, A.J.: Fast, Flexible, and Minimal CTL Synthesis via SMT. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 136–156. Springer International Publishing, Cham (2016)
29. Luckow, K.S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: JDart: A Dynamic Symbolic Analysis Framework. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 442–459. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_26, https://doi.org/10.1007/978-3-662-49674-9_26
30. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Jr., W.A.H., Somenzi, F. (eds.) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8–12, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1, https://doi.org/10.1007/978-3-540-45069-6_1
31. de Moura, L.M., Bjørner, N.: Proofs and Refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008), <http://ceur-ws.org/Vol-418/paper10.pdf>
32. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, 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. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24

33. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-Specific Proof Steps Witnessing Correctness of SMT Executions. In: 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021. pp. 541–546. IEEE (2021). <https://doi.org/10.1109/DAC18074.2021.9586272>, <https://doi.org/10.1109/DAC18074.2021.9586272>
34. Ozdemir, A., Niemetz, A., Preiner, M., Zohar, Y., Barrett, C.W.: DRAT-based Bit-Vector Proofs in CVC4. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 298–305. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_21, https://doi.org/10.1007/978-3-030-24258-9_21
35. Selman, B., Kautz, H.A.: Knowledge Compilation using Horn Approximations. In: Dean, T.L., McKeown, K.R. (eds.) Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 2. pp. 904–909. AAAI Press / The MIT Press (1991), <http://www.aaai.org/Library/AAAI/1991/aaai91-140.php>
36. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. *Formal Methods Syst. Des.* **42**(1), 91–118 (2013). <https://doi.org/10.1007/s10703-012-0163-3>, <https://doi.org/10.1007/s10703-012-0163-3>
37. Wetzler, N., Heule, M., Jr., W.A.H.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 422–429. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31, https://doi.org/10.1007/978-3-319-09284-3_31
38. Wetzler, N., Heule, M.J.H., Hunt, W.A.: Mechanical Verification of SAT Refutations with Extended Resolution. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving. pp. 229–244. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
39. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany. pp. 10880–10885. IEEE Computer Society (2003). <https://doi.org/10.1109/DATE.2003.10014>, <http://doi.ieeecomputersociety.org/10.1109/DATE.2003.10014>

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.





Z3-NOODLER: An Automata-based String Solver

Yu-Fang Chen¹, David Chocholatý², Vojtěch Havlena²,
Lukáš Holík²(✉), Ondřej Lengál², and Juraj Síč²

¹ Institute of Information Science, Academia Sinica, Taipei, Taiwan

² Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

holik@fit.vutbr.cz

Abstract. Z3-Noodler is a fork of Z3 that replaces its string theory solver with a custom solver implementing the recently introduced stabilization-based algorithm for solving word equations with regular constraints. An extensive experimental evaluation shows that Z3-Noodler is a fully-fledged solver that can compete with state-of-the-art solvers, surpassing them by far on many benchmarks. Moreover, it is often complementary to other solvers, making it a suitable choice as a candidate to a solver portfolio.

1 Introduction

Recently, many tools for solving string constraints have been developed, motivated mainly by techniques for finding security vulnerabilities such as SQL injection or cross-site scripting (XSS) in web applications [34,35,36]. String solving has also found its applications in, e.g., analysis of access user policies in Amazon Web Services [26,8,39] or smart contracts [7]. Solvers for string constraints are usually implemented as string theory solvers inside SMT solvers, such as *cvc5* [9] or Z3 [31], allowing combination with other theories, most commonly the theory of integers for string lengths. Other well known string solvers include Z3str3RE [13,12], Z3-Trail [1], Z3str4 [30], OS-TRICH [19], and others.

In this paper, we present Z3-Noodler 1.0.0 [47], a fork of Z3 4.12.2 where the string theory solver is replaced with the *stabilization-based procedure* for solving string (dis)equations with regular and length constraints [14,20]. The procedure makes heavy use of *nondeterministic finite automata* (NFAs) and operations over them, for which we use the efficient *Mat a* library for NFAs [23,29].

The presented version implements multiple improvements over a previous Z3-Noodler prototype from [20]. Firstly, it extends the support for string predicates from the SMT-LIB string theory standard [11] by (1) applying smarter and more specific axiom saturation and (2) adding support for their solving inside the decision procedure (e.g., for the `¬contains` predicate). It also implements various optimizations (e.g., for regular constraints handling) and other decision procedures, e.g., the *Nielsen transformation* [32] for quadratic equations and a procedure for regular language (dis)equations; moreover, we added heuristics for choosing the best decision procedure to use.

We compared Z3-Noodler with other string solvers on standard SMT-LIB benchmarks [10,42,43]. The results indicate that Z3-Noodler is competitive, superior especially on benchmarks containing mostly regular constraints and word (dis)equations, and that the improvements since [20] had a large impact on the number of solved instances as well as its overall performance.

2 ARCHITECTURE

Z3-Noodler replaces the string theory solver in the DPLL(T)-based SMT solver Z3 [31] (version 4.12.2) with our string solver Noodler [14], which is based on the *stabilization algorithm* (cf. Section 3). DPLL(T)-based solvers in general combine a SAT solver providing satisfying assignments to the Boolean skeleton of a formula with multiple theory solvers for checking conjunctions of theory literals.

Z3-Noodler still uses the infrastructure of Z3, most importantly the parser, string theory rewriter and the *linear integer arithmetic* (LIA) solver. The Z3 parser takes formulae in the SMT-LIB format [10], where Z3-Noodler can handle nearly all predicates/functions (such as `substr`, `len`, `at`, `replace`, regular membership, word equations, etc.) in the string theory as defined by SMT-LIB [11].

Even though we do use the string theory rewriter of Z3, we disabled those rewritings that do not benefit our core string solver. For instance, we removed rules that rewrite regular membership constraints to other types of constraints since solving regular constraints and word equations using our stabilization-based approach is efficient.

The interaction of the Noodler solver with Z3 is shown in Fig. 1 and works as follows. Upon receiving a satisfying Boolean assignment from the SAT solver (①), we first remove irrelevant assignments (using Z3’s relevancy propagation), which allows us to work with smaller instances and return more general theory lemmas. A theory assignment obtained from the Boolean assignment consists of string (dis)equations, regular constraints, and, possibly, predicates that were not *axiom-saturated* before (cf. Section 3).

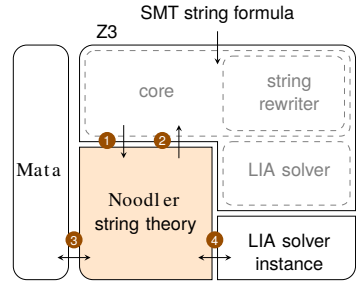


Fig. 1: Architecture of Z3-Noodler

The core Noodler string decision procedure then reduces the conjunction of string literals to a LIA constraint over string lengths, and returns it to Z3 as a theory lemma (②), to be solved together with the rest of the input arithmetic constraints by Z3’s internal LIA solver. Noodler implements a couple of decision procedures (discussed in Section 3), heavily employing the Mata automata library (version 0.109.0) [29] (③). As an optimization of the theory lemma generation, when the string constraint reduces into a disjunction of LIA length constraints, we check the satisfiability of individual disjuncts (generated lazily on demand) separately in order to get a positive answer as soon as possible. For testing the disjuncts, the current solver context is cloned and queried about satisfiability of the LIA constraint conjoined with the disjunct (④).

3 STRING THEORY CORE

In this section, we provide details about Z3-Noodler’s string theory implementation, including initial axiom saturation, preprocessing, the core procedure, and limitations.

Axiom Saturation. In order to best utilize the power of Z3’s internal LIA solver during the generation of a satisfiable assignment, we saturate the input formula with length-aware

theory axioms and axioms for string predicates (this happens during Z3's processing of the input formula, before the main SAT solver starts generating assignments). We can then avoid checking SAT assignments that trivially violate length conditions. Most importantly, we add length axioms $\text{len}(t_1) \geq 0$, $\text{len}(t_1.t_2) = \text{len}(t_1) + \text{len}(t_2)$ where t_1, t_2 are arbitrary string terms, and $\text{len}(t_1) = \text{len}(t_2)$ for the word equation $t_1 = t_2$.

Moreover, for string functions/predicates, Noodler saturates the original formula with an equivalent formula composed of word (dis)equations and length/regular constraints, which are more suitable for our core procedure (e.g., for $\neg \text{contains}(s, \text{"abc"})$ in the input formula, we add the regular constraint $s \notin \Sigma^* \text{abc} \Sigma^*$). We use different saturation rules for instances of predicates with concrete values. For instance, for $\text{substr}(s, 4, 1)$, we add just the term $\text{at}(s, 4)$. On the other hand, for $\text{substr}(s, t_i, t_j)$, where s is a string term and t_i, t_j are general integer terms (possibly containing variables), we need to add a more general formula talking about the prefix and suffix of s of given lengths. The original predicate occurrence is then removed from received assignments by Noodler (Z3 does not allow to remove parts of the original formula).

Decision Procedures. Z3-Noodler's string theory core contains several complementary decision procedures. The main one is the *stabilization-based* algorithm for solving word equations with regular constraints introduced in [14] and later extended with efficient handling of length constraints and disequations [20]. The stabilization-based algorithm starts, for every string variable, with an NFA encoding regular constraints on the variable and iteratively refines the NFA according to the word equations until the stability condition is achieved. The stability condition holds when, for every word equation, the language of the left-hand side (obtained as the language of the concatenation of NFAs for variables and string literals) equals the language of the right-hand side. When stability is achieved, length constraints of the solutions are generated and passed to the LIA solver. The algorithm is complete for the *chain-free* [5] combinations of equations, regular and length constraints, together with unrestricted disequations, making it the largest known decidable fragment of these types of constraints.

The stabilization-based decision procedure starts by inductively converting the initial regular constraints into NFAs. During the construction, we utilize eager simulation-based reduction [16,17] with on-demand determinization and minimization.

For an efficient handling of *quadratic* equations (systems of equations with at most two occurrences of each variable) with lengths, Noodler implements a decision procedure based on the *Nielsen transformation* [32]. The algorithm constructs a graph corresponding to the system and reasons about it to determine if the input formula is satisfiable or not [38,22]. If the system contains length variables, we also create a counter automaton corresponding to the Nielsen graph (in a similar way as in [28]). In the subsequent step, we contract edges, saturating the set of self-loops and, finally, we iteratively generate flat counter sub-automata (a flat counter automaton only allows cycles that are self-loops), which are later transformed into LIA formulae describing lengths of all possible solutions.

In order to solve *(dis)equations of regular expressions*, we reduce the problem to reasoning about the corresponding NFAs (similarly as for regular constraints handling). In particular, we use efficient NFA equivalence and universality checking from Mata, which implements advanced antichain-based algorithms [46,6].

Preprocessing. Each decision procedure employs a sequence of preprocessing rules transforming the string constraint to a more suitable form. Our portfolio of rules includes transformations reducing the number of equations by a conversion to regular constraints, propagating epsilons and variables over equations, underapproximation rules, and rules reducing the number of disequations (cf. [20]). On top of that, Z3-Noodler employs information about length-equivalent variables allowing to infer simpler constraints (e.g., for $xy = zw$ with $\text{len}(x) = \text{len}(z)$, we can infer $y = w$). Z3-Noodler also checks for simple unsatisfiable patterns for early termination. A sequence of preprocessing rules is composed for each of the decision procedures differently, maximizing their strengths.

Supported String Predicates and Limitations. Z3-Noodler currently supports handling of basic string predicates `replace`, `substr`, `at`, `indexof`, `prefix`, `suffix`, `contains`, and a limited support for `¬contains`. From the set of extended constraints, the core solver currently does not support the `replace_all` function (and variants of replacement based on regular expressions) and `to/from_int` conversions. The decision procedures used in Z3-Noodler make it complete for the chain-free fragment with unbounded disequations and regular constraints [20], and quadratic equations. Outside this fragment, our theory core is sound but incomplete.

4 Experiments

Tools and environment. We compared Z3-Noodler with the following state-of-the-art tools: `cvc5` [9] (version 1.0.8), `Z3` [31] (version 4.12.2), `Z3str3RE` [13,12], `Z3str4` [30], `OSTRICH` [19]³, and Z3-Noodler^{pr} (version 0.1.0 used in [20]). We did not compare with Z3-Trail [2] as it is no longer under active development and gives incorrect results on newer benchmarks. The experiments were executed on a workstation with an Intel Xeon Silver 4314 CPU @ 2.4 GHz with 128 GiB of RAM running Debian GNU/Linux. The timeout was set to 120 s, memory limit was set to 8 GiB.

Benchmarks. The benchmarks come from the SMT-LIB [10] repository, specifically categories `QF_S` [42] and `QF_SLIA` [43]. These benchmarks were also used in SMT-COMP’23 [41], in which Z3-Noodler participated (version 0.2.0). As Z3-Noodler does not support `to/from_int` conversions and `replace_all`-like predicates, we excluded formulae whose satisfiability checking needs their support. Based on the occurrences of different kinds of constraints, we divide the benchmarks into three groups:

REGEX This category contains formulae with dominating regular membership and length constraints. It consists of `AutomatArk` [13], `Denghang`, `StringFuzz` [15], and `Sygyus-qgen` benchmark sets. We excluded 1,568 formulae from `StringFuzz` that require support of the `to_int` predicate.

EQUATIONS The formulae in this category consist mostly of word equations with length constraints and a small amount of other predicates. It contains `Kaluza` [40,27], `Kepler` [25], `Norn` [3,4], `Slent` [44], `Slog` [45], `Webapp`, and `Woorpje` [24] benchmark sets. We excluded 414 formulae from `Webapp` that require support of `replace_all`, `replace_re`, and `replace_re_all` predicates.

³ Latest commit 70d01e2d2, run with `-portfolio=strings` option.

Table 1: Results of experiments on all benchmark sets. For each tool and benchmark set (as well as whole groups under Σ), we give the number of unsolved instances. Results for tools with the highest number of solved instances are in **bold**. Numbers with * contain also incorrect results.

	Regex						Equations							Predicates-small					PyEx
	Aut	Den	StrFuzz	Syg	Σ		Kal	Kep	Norn	Slent	Slog	Web	Woo	Σ	StrInt	Leet	StrSm	Σ	
Included	15,995	999	10,050	343	27,387		19,432	587	1,027	1,128	1,976	267	809	25,226	11,669	2,652	1,670	15,991	23,845
Unsupported	0	0	1,568	0	1,568		0	0	0	0	0	414	0	414	5,299	0	210	5,509	0
Z3-Noodler	62	0	0	0	62		259	4	0	5	0	0	243	511	4	4	55	63	4,424
cvc5	94	18	1037	0	1149		0	240	85	22	0	40	54	441	0	0	4	4	34
Z3	113	118	340	0	571		164	313	124	74	71	61	25	832	4	0	32	36	1,071
Z3str4	60	4	27	0	91		174	254	73	73	16	62	78	730	5	4	37	46	570
OSTRICH	55	15	229	0	299		288	387	1	130	7	65	53	931	37	26	*106	*169	12,290
Z3str3RE	66	27	*143	1	*237		*144	311	133	87	55	*104	*118	*952	64	192	*179	*435	17,764
Z3-Noodler ^{pr}	86	1	*1,014	0	*1,101		508	575	0	6	0	*3	256	*1,348	40	29	*493	*562	*13,362

PREDICATES-small. Although Z3-Noodler focuses mainly on word equations with length and regular constraints, the evaluation includes also a group consisting of smaller formulae that use string predicates such as `substr`, `at`, `contains`, etc. It is formed from FullStrInt, LeetCode, and StrSmallRw [33] benchmark sets. We removed 5,509 formulae containing the `to/from_int` predicates from FullStrInt and StrSmallRw.

We also consider the PyEx [37] benchmark, which we do not put into any of these groups, as it contains large formulae with complex predicates (`substr`, `contains`, etc.). We note that we omit the small Transducer+ [18] benchmark because it contains exclusively formulae with `replace.all`.

Results. We show the number of unsolved instances for each benchmark and tool (as well as whole groups) in Table 1. Some tools gave incorrect results (determined by comparing to the output of `cvc5` and Z3) for some benchmarks. Usually, this was less than 10 instances, except for Z3str3RE on StringFuzz and StrSmallRw (50 and 12 incorrect results respectively) and Z3-Noodler^{pr} on StrSmallRw (218 incorrect results). Table 2 then shows the average run times and their standard deviations for solved instances for each category and tool.

The results show that Z3-Noodler outperforms other tools on the **Regex** group (in particular on Denghang, StringFuzz, and Sygus-qgen) both in the number of solved instances and the average run time. Only on AutomatArk it cannot solve the most formulae (but it solves only 7 less than the winner OSTRICH, while being much faster).

On the **Equations** group, Z3-Noodler also outperforms other tools on most of the benchmarks. In particular on Kepler, Norn, Slent, Slog, and Webapp. On Kaluza, it is outperformed by other tools, but it still solves the vast majority of formulae. Z3-Noodler has worse performance on Woopje, which seems to be a synthetic benchmark generated to showcase the strength of a specialized algorithm [24] (this benchmark is the reason for Z3-Noodler taking the second place in the whole group). With 0.11 s, Z3-Noodler and `cvc5` have the lowest average run time.

Table 2: Average run times (in seconds) of solved instances and their standard deviations.

	Reg		Eq		Pred	
	avg	std	avg	std	avg	std
Z3-Noodler	0.11	1.35	0.11	2.13	0.11	2.16
cvc5	1.17	8.51	0.11	2.15	0.03	0.15
Z3	1.92	9.71	0.18	2.83	0.04	0.42
Z3str4	0.35	2.00	0.25	3.40	0.02	0.31
OSTRICH	4.29	8.67	4.28	9.28	12.71	15.08
Z3str3RE	0.31	3.28	0.13	2.72	0.01	0.08
Z3-Noodler ^{pr}	0.27	2.86	0.12	2.93	0.09	1.69

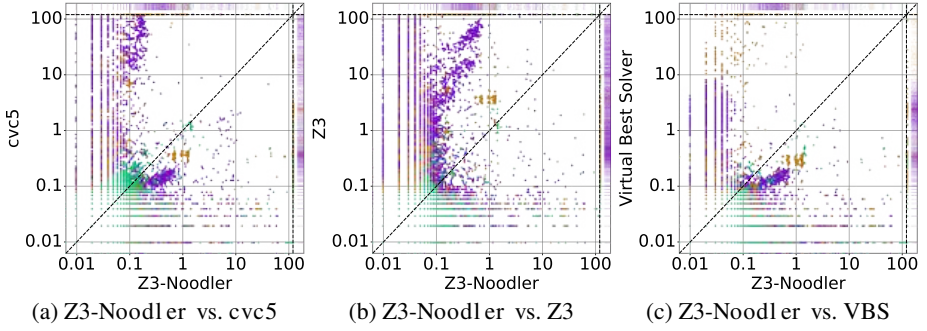


Fig. 2: Comparison of Z3-Noodler with cvc5, Z3, and the virtual best solver (VBS). Times are in seconds, axes are logarithmic. Dashed lines represent timeouts (120 s). Colours distinguish groups: **Regex**, **Equations**, and **Predicates-small**.

The winner of **Predicates-small** is cvc5. In particular, on FullStrInt and LeetCode the difference with Z3-Noodler is equally 4 instances and on StrSmallRw the difference is 51 cases. The average time of Z3-Noodler is also a bit higher, with 0.11 s for Z3-Noodler compared to the 0.03 s for cvc5. Similarly, Z3-Noodler is outperformed by cvc5, Z3, and Z3str4 on PyEx. Indeed, we have not optimized Z3-Noodler for formulae with large numbers of predicates yet. The results of Z3-Noodler could, however, be further improved by proper axiom saturation for predicates or lazy predicate evaluation.

In Fig. 2 we show scatter plots comparing running time of Z3-Noodler with cvc5, Z3, and virtual best solver (VBS; a solver that takes the best result from all tools other than Z3-Noodler) on all three benchmark groups. The plots show that Z3-Noodler outperforms the competitors on a vast number of instances, in many cases being complementary to them. To validate this claim, we also checked how different solvers contribute to a portfolio. That is, we took the VBS including Z3-Noodler (VBS⁺) and then checked how well the portfolio works without each of the solvers. Table 3 shows the results on the **Regex** and **Equations** groups (we omit **Predicates-small**, where Z3-Noodler does not help the portfolio). The results show that on the two groups, Z3-Noodler is the most valuable solver in the portfolio. We also include results on the small portfolio of Z3 and cvc5 (with and without Z3-Noodler) showing that, on the two groups, using just these three solvers is almost as good as using the whole portfolio of all solvers.

Table 3: Evaluating solver contribution to a portfolio. Times are in seconds.

	Regex		Equations	
	Unsolved	Time	Unsolved	Time
VBS⁺	1	427	19	1,304
VBS ⁺ - Z3-Noodler	1	2,914	131	6,830
VBS ⁺ - cvc5	1	549	145	1,401
VBS ⁺ - Z3	1	430	29	1,579
VBS ⁺ - Z3str4	1	473	19	1,416
VBS ⁺ - OSTRICH	1	427	21	1,270
VBS ⁺ - Z3str3RE	1	510	20	1,307
cvc5 + Z3 + Z3-Noodler	1	608	22	1,471
cvc5 + Z3	278	27,916	303	2,805

Comparing with the older version Z3-Noodler^{pr} from [20], we can see that there is a significant improvement in most benchmarks, most significantly in AutomatArk, StringFuzz, Kepler, StrSmallRw, and Kaluza. We note that adding more complicated algorithm selection strategies significantly improved the overall performance of Z3-

Noodler, but, on the other hand, decreased the performance on Kaluza (cf. [20]). Better results in AutomatArk and StringFuzz stem from the improvements in MATA and from heuristics tailored for regular expressions handling. Including Nielsen's algorithm [32] has the largest impact on the Kepler benchmark. The improvement on predicate-intensive benchmarks is caused by optimizations in axiom saturation for predicates. The older version also had multiple bugs that have been fixed in the current version.

Acknowledgments

This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, and the FIT BUT internal project FIT-S-23-8151.

Data Availability Statement

An environment with the tools and data used for the experimental evaluation in the current study is available at [21].

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Dolby, J., Janků, P., Lin, H., Holík, L., Wu, W.: Efficient handling of string-number conversion. In: Proc. of PLDI'20. pp. 943–957. ACM (2020). <https://doi.org/10.1145/3385412>, <https://doi.org/10.1145/3385412>
2. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–5. IEEE (2018). <https://doi.org/10.23919/FMCAD.2018.8602997>, <https://doi.org/10.23919/FMCAD.2018.8602997>
3. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) 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. Lecture Notes in Computer Science, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_10, https://doi.org/10.1007/978-3-319-08867-9_10
4. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 462–469. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_29, https://doi.org/10.1007/978-3-319-21690-4_29
5. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 277–293. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_16, https://doi.org/10.1007/978-3-030-31784-3_16

6. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: TACAS'10. LNCS, vol. 6015, pp. 158–174. Springer (2010)
7. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 325–338. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_16, https://doi.org/10.1007/978-3-031-13185-1_16
8. Backes, J., Bolognani, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for aws access policies using smt. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018). <https://doi.org/10.23919/FMCAD.2018.8602994>
9. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
10. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
11. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB): Strings. <https://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml> (2023)
12. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: Towards more efficient methods for solving regular-expression heavy string constraints. *Theor. Comput. Sci.* **943**, 50–72 (2023). <https://doi.org/10.1016/j.tcs.2022.12.009>, <https://doi.org/10.1016/j.tcs.2022.12.009>
13. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 289–312. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_14, https://doi.org/10.1007/978-3-030-81688-9_14
14. Blahoudek, F., Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Word equations in synergy with regular constraints. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. pp. 403–423. Springer International Publishing, Cham (2023)
15. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: StringFuzz: A fuzzer for string solvers. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 45–51. Springer International Publishing, Cham (2018)
16. Bustan, D., Grumberg, O.: Simulation based minimization. In: Proceedings of CADE-17. LNCS, vol. 1831, pp. 255–270. Springer (2000)
17. Cécé, G.: Foundation for a series of efficient simulation algorithms. In: LICS'17. pp. 1–12. IEEE Computer Society (2017)
18. Chen, T., Hague, M., He, J., Hu, D., Lin, A.W., Rümmer, P., Wu, Z.: A decision procedure for path feasibility of string manipulating programs with integer data type. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12302, pp. 325–342. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6_18, https://doi.org/10.1007/978-3-030-59152-6_18
19. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* **3**(POPL), 49:1–49:30 (2019). <https://doi.org/10.1145/3290362>, <https://doi.org/10.1145/3290362>

20. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Solving string constraints with lengths by stabilization. *Proc. ACM Program. Lang.* 7(OOPSLA2) (oct 2023). <https://doi.org/10.1145/3622872>
21. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-Noodler: An automata-based string solver (Oct 2023). <https://doi.org/10.5281/zenodo.10041441>, <https://doi.org/10.5281/zenodo.10041441>
22. Chen, Y.F., Havlena, V., Lengál, O., Turrini, A.: A symbolic algorithm for the case-split rule in solving word constraints with extensions. *Journal of Systems and Software* **201**, 111673 (2023). <https://doi.org/10.1016/j.jss.2023.111673>, <https://www.sciencedirect.com/science/article/pii/S0164121223000687>
23. Chocholatý, D., Fiedor, T., Havlena, V., Holík, L., Hruška, M., Lengál, O., Síč, J.: Mata: A fast and simple finite automata library. In: *Proc. of TACAS'24. LNCS*, Springer (2024)
24. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R.M., Potapov, I. (eds.) *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 11674, pp. 93–106. Springer (2019). https://doi.org/10.1007/978-3-030-30806-3_8, https://doi.org/10.1007/978-3-030-30806-3_8
25. Le, Q.L., He, M.: A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In: Ryu, S. (ed.) *Programming Languages and Systems*. pp. 350–372. Springer International Publishing, Cham (2018)
26. Liana Hadarean: String solving at Amazon. <https://mosca19.github.io/program/index.html> (2019), presented at MOSCA'19
27. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. *Formal Methods in System Design* **48**(3), 206–234 (2016)
28. Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility. In: *Automated Technology for Verification and Analysis*. pp. 352–369. Springer International Publishing, Cham (2018)
29. Mata: An efficient automata library (2023), <https://github.com/VeriFIT/mata>
30. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Lecture Notes in Computer Science*, vol. 13047, pp. 389–406. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_21, https://doi.org/10.1007/978-3-030-90870-6_21
31. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: *TACAS'08. LNCS*, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
32. Nielsen, J.: Die isomorphismen der allgemeinen, unendlichen gruppe mit zwei erzeugenden. *Mathematische Annalen* **78**(1), 385–397 (1917)
33. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2019*. pp. 279–297. Springer International Publishing, Cham (2019)
34. OWASP: Top 10. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf (2013)
35. OWASP: Top 10. <https://owasp.org/www-project-top-ten/2017/> (2017)
36. OWASP: Top 10. <https://owasp.org/Top10/> (2021)
37. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kunčák, V. (eds.) *Computer Aided Verification*. pp. 453–474. Springer International Publishing, Cham (2017)
38. Robson, J.M., Diekert, V.: On quadratic word equations. In: *Annual Symposium on Theoretical Aspects of Computer Science*. pp. 217–226. Springer (1999)

39. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vize, Y. (eds.) *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 13371, pp. 3–18. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_1, https://doi.org/10.1007/978-3-031-13185-1_1
40. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: Kaluza web site (2023), <https://webblaze.cs.berkeley.edu/2010/kaluza/>
41. SMT-COMP'23: <https://smt-comp.github.io/2023/> (2023)
42. SMT-LIB: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S (2023)
43. SMT-LIB: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA (2023)
44. Wang, H.E., Chen, S.Y., Yu, F., Jiang, J.H.R.: A symbolic model checking approach to the analysis of string and length constraints. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, p. 623–633. ASE 2018, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3238147.3238189>, <https://doi.org/10.1145/3238147.3238189>
45. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: *CAV'16. LNCS*, vol. 9779, pp. 241–260. Springer (2016)
46. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: *CAV'06. LNCS*, vol. 4144, pp. 17–30. Springer (2006)
47. Z3-Noodler: Automata-based string solver (2023), <https://github.com/VeriFIT/z3-noodler>

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.





TaSSAT: Transfer and Share SAT*

Md Solimul Chowdhury^(✉), Cayden R. Codel, and Marijn J. H. Heule

Carnegie Mellon University, Pittsburgh, PA, USA

{mdsolimc, ccodel, mheule}@cs.cmu.edu

Abstract. We present TaSSAT, a powerful local search SAT solver that effectively solves hard combinatorial problems. Its unique approach of transferring clause weights in local minima enhances its efficiency in solving problem instances. Since it is implemented on top of YaSAT, TaSSAT benefits from practical techniques such as restart strategies and thread parallelization. Our implementation includes a parallel version that shares data structures across threads, leading to a significant reduction in memory usage. Our experiments demonstrate that TaSSAT outperforms similar solvers on a vast set of SAT competition benchmarks. Notably, with the parallel configuration of TaSSAT, we improve lower bounds for several van der Waerden numbers.

Keywords: Local Search for SAT · Weight Transfer · Memory Efficiency

1 Introduction

The SAT problem asks if there exists a satisfying truth assignment for a given formula in propositional logic. SAT is known to be intractable [11], but modern SAT solvers, particularly conflict-driven clause learning (CDCL) solvers, have made significant progress in solving large formulas from various application domains. When it comes to combinatorial problems, stochastic local search (SLS) solvers are often more effective than CDCL. Because SLS and CDCL solvers have complementary strengths, some SAT solvers like Kissat [7] and CryptoMiniSAT [17] combine SLS and CDCL techniques, and SLS methods play a key role in shaping the capabilities of modern SAT solvers.

SLS solvers explore truth assignments by flipping the truth value of individual variables until a solution is found or until timeout. The solver generally tries to flip variables that will minimize the number of falsified clauses. When a solver determines that no variable flip will lead to an improvement according to some heuristic or metric, it has reached a local minimum.

To escape local minima, the solver can either make random flips or adjust its internal state until improvement is possible. Despite being an effective family of algorithms for escaping local minima, Dynamic Local Search (DLS) has attracted

* The authors were supported by NSF grant CCF-2229099. Md Solimul Chowdhury was partially supported by a NSERC Postdoctoral Fellowship.

limited attention in the recent years. DLS algorithms assign weights to clauses, search to find a solution by minimizing the total amount of weight held by falsified clauses, and adjust these weights in local minima as a means of escaping them.

The tool we present in this paper is ultimately based on DDFW [16] (divide and distribute fixed weights), a DLS algorithm that dynamically transfers weight from satisfied to falsified clauses along neighborhood relationships in local minima. DDFW is remarkably effective at solving hard combinatorial problems, such as matrix multiplication [14], graph coloring [13], edge matching [12], the coloring of the Pythagorean triples [15], and finding bounds for van der Waerden numbers [3]. Notably, DDFW solves satisfiable instances of the Pythagorean triples problem in under a minute, whereas CDCL solvers take CPU years.

In this paper, we introduce Transfer and Share SAT (TaSSAT), a novel parallel SLS solver. TaSSAT implements LiWeT, a simplification of the algorithm from our recent work [10] modifying DDFW. Our implementation of TaSSAT is built on top of a leading SLS solver YaSAT [5], and it adds two new features. First, it incorporates the weight-transfer methods from LiWeT, leading to more efficient solving. Specifically, a new weight-transfer parameter allows TaSSAT to shift more clause weight in local minima, enhancing its adaptability during the search. Second, TaSSAT’s parallel mode shares data structures among threads to reduce its memory footprint by up to 80%.

Our results show that TaSSAT substantially outperforms YaSAT on an extensive benchmark set of 5355 anniversary instances from the 2022 SAT Competition. Further, TaSSAT’s parallel version improves the lower bounds for nine van der Waerden numbers, surpassing prior work by Ahmed et al. [3] that used 29 algorithms (including DDFW) and extensive parallelization. Our results demonstrate the clear algorithmic and practical improvements of TaSSAT.

2 Preliminaries

A SAT formula in conjunctive normal form (CNF) is a conjunction of clauses, each of which is a disjunction of literals (Boolean variables or their negations). A clause C is satisfied by a truth assignment α if α satisfies at least one of its literals, and is otherwise falsified. A formula F is satisfied by α when all of its clauses are. Clauses C and D are *neighbors* if they share a common literal.

In DLS, clauses are assigned weights, denoted as $W : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$, representing the cost of leaving a clause falsified. The total weight of the falsified clauses is the *falsified weight*. Variables that reduce the falsified weight when flipped are called *weight-reducing variables*, while those that do not impact the falsified weight when flipped are called *sideways variables*.

DDFW starts with a random initial truth assignment and sets all clause weights to parameter w_0 ($w_0 = 8$ in the original paper [16]). It then flips weight-reducing variables until none remain. Upon reaching a local minimum, DDFW randomly chooses between making a sideways flip (if possible, and with a 15% chance) or entering the weight transfer phase. During weight transfer, each falsi-

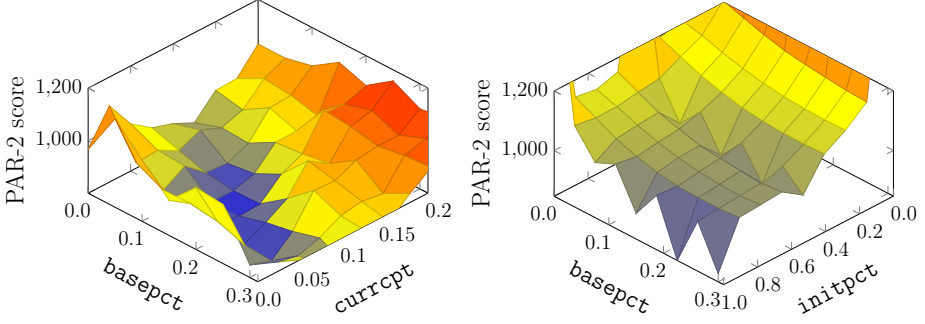


Fig. 1: PAR-2 scores for parameter searches on `initpct`, `basepct`, and `currpct`. The plots are oriented to best show the performance trends, so the axes vary.

fied clause receives a fixed weight from a maximum-weight satisfied neighbor C_S (except for 1% of the time, when a random satisfied clause is chosen instead). The amount of weight transferred from C_S depends on its weight: if $W(C_S) > w_0$, then a weight of 2 is taken; otherwise, a weight of 1 is taken.

3 LiWeT: The Linear Weight Transfer Algorithm

TaSSAT takes ideas from DDFW and distills them into an algorithm called LiWeT (Linear Weight Transfer), which is a simplification of our prior work [10]. LiWeT uses a novel linear weight transfer rule to determine how much weight to move in local minima. The rule takes three parameters: `currpct`, a multiplier on the current clause’s weight; `basepct`, a multiplier on the initial weight w_0 ; and `initpct`, a multiplier for clauses with exactly w_0 weight. For most clauses C_s , the amount of weight that is transferred is `currpct` · $W(C_S)$ + `basepct` · w_0 . For clauses with $W(C_S) = w_0$, the amount taken is `initpct` · w_0 . As a result, `initpct` controls how much weight is initially taken from a clause.

The weight transfer rule offers two key advantages. First, the use of floating-point parameters rapidly establishes distinct weights for clauses, eliminating the need for tie-breaking near local minima and, consequently, explicit sideways flips. Second, the `initpct` parameter enables LiWeT to release a larger proportion of the total clause weight, enhancing its adaptability to challenging formulas. In DDFW and LiWeT, maximum-weight neighbors are selected for each falsified clause within local minima. Clauses with weights less than w_0 are unlikely to contribute more weight, artificially reducing the total amount of weight LiWeT can move around. The `initpct` parameter prevents this from happening.

LiWeT differs from DDFW in one other respect: in local minima, it increases the probability of choosing a randomly satisfied clause, rather than a maximum-weight neighbor, to 10%. We found that this improves overall performance.

Algorithm 1 shows LiWeT’s pseudocode.

Algorithm 1: The LiWeT algorithm

Input: CNF formula F , w_0 , **initpct**, **basepct**, **currpct**
Output: Satisfiability of F

```

1  $W(C) \leftarrow w_0$  for all  $C \in F$ 
2  $\alpha \leftarrow$  random truth assignment on the variables in  $F$ 
3 for 1 to MAXFLIPS do
4   if  $\alpha$  satisfies  $F$  then return “SAT”
5   else
6     if a weight reducing variable is available then
7       flip the variable that reduces the falsified weight the most
8     else
9       foreach clause  $C \in F$  falsified under  $\alpha$  do
10         $C_S \leftarrow$  select a satisfied clause
11        if  $W(C_S) = w_0$  then     $w \leftarrow \text{initpct} \cdot w_0$ 
12        else                       $w \leftarrow \text{currpct} \cdot W(C_S) + \text{basepct} \cdot w_0$ 
13        transfer  $w$  from  $C_S$  to  $C$ 
14 return “No SAT”

```

To determine the effect of the three parameters, we conducted parameter searches across them. We ranged **basepct** $\in [0, 0.3]$, **currpct** $\in [0, 0.2]$, and **initpct** $\in [0, 1.0]$ with increments of 0.1, 0.05 and 0.2, respectively. Our searches were done on a combined 168 instances from the 2019 SAT Race and the 2021 and 2022 SAT competitions, each with a 900-second timeout. We picked these instances because they were solved by previous versions of LiWeT and DDFW, and thus were less likely to result in timeout.

Figure 1 shows the PAR-2 scores for two parameter searches, where a lower score indicates better performance.¹ The left plot shows that TaSSAT performs better with higher values of both **basepct** and **currpct** when **initpct** = 1. The optimal configuration is (**basepct**, **currpct**) = (0.175, 0.075). The right plot shows that LiWeT performs best when **initpct** = 1 for any **basepct** value when **currpct** = 0. This suggests that taking all weight from satisfied clauses early in the search is crucial for better performance. We ran all subsequent TaSSAT experiments with (**initpct**, **basepct**, **currpct**) = (1, 0.175, 0.075).

We conclude this section by outlining the distinctions between the algorithm presented in [10] and LiWeT, underscoring the simplifications introduced in the latter compared to the former. Compared to the algorithm from our previous work [10], LiWeT has two fewer parameters. Previously, the algorithm used two pairs of (a, c) parameters to transfer $a * W(C_S) + c$ weight from satisfied clauses C_S in local minima. One pair of (a, c) values was used when $W(C_S) > w_0$, and the other for when $W(C_S) = w_0$. In LiWeT, we replaced the second pair with **initpct**. Then based on the observation in the right plot of Figure 1, we set

¹ The PAR-2 score is defined as the average solving time, with twice the timeout as the time for unsolved instances.

`initpct` to 1 for performance reasons. This adjustment eliminates `initpct` from line 11 of Algorithm 1, transforming it into a two-parameter algorithm.

Another simplification was the removal of sideways variable flips from LiWeT. DDFW and previous versions of our algorithm would flip sideways variables, but we found that they rarely occurred with floating-point weights, and refusing to flip them didn't affect performance. Notably, these simplifications enhance the algorithmic power of LiWeT over the previous algorithm, which we demonstrate in section 5.

4 Implementation of TaSSAT and PaSSAT

We implemented TaSSAT on top of YalSAT [6], a state-of-the-art SLS solver that implements the ProbSAT algorithm [4]. As a result, our implementation benefits from the practical techniques present in YalSAT, including restart techniques. Our TaSSAT implementation² includes a parallel version, called PaSSAT, that improves the memory management of the parallel version of YalSAT.

Because LiWeT is computationally expensive when there are a higher number of falsified clauses, TaSSAT has an optional mode to run ProbSAT until the number of falsified clauses drops beneath a dynamically computed threshold based on the formula's size, at which point it resumes LiWeT. By default, we ran TaSSAT with this option disabled in our experiments, but we enabled it for the van der Waerden experiments.

We also improve on the parallel features in YalSAT. The main issue in the parallel version of YalSAT was that the formula data structures were not shared. As a result, each thread had to independently parse, store, and simplify the input formula, resulting in redundant computation and a bloated memory footprint. We solved this problem in PaSSAT by nominating a primary thread to parse and simplify the formula and to allocate the core data structures. Once the primary thread finishes, it hands solving off to the secondary threads, which can then jointly refer to the shared data structures.

5 Evaluation

We now present our experimental results³ of TaSSAT against similar algorithms. Our baseline solvers are the original YalSAT (YalSAT-Prob); our DDFW-inspired, YalSAT-based solver from previous work [10] (YalSAT-Lin); a YalSAT-based implementation of DDFW (YalSAT-DDFW); and the UBCSAT implementation of DDFW (UBCSAT-DDFW). We include two DDFW implementations to check that the YalSAT version performs similarly to the UBCSAT one, despite being implemented with a different base solver.

We ran these four solvers on two benchmark sets: a set of 5355 instances from the 2022 SAT Competition's anniversary track (the `anni` set) [1] covering instances from the previous 20 years of competition, and a set of nine van

² TaSSAT source code is available at <https://github.com/solimul/tassat>.

³ Details are available at https://github.com/solimul/TACAS-24-solve_details.

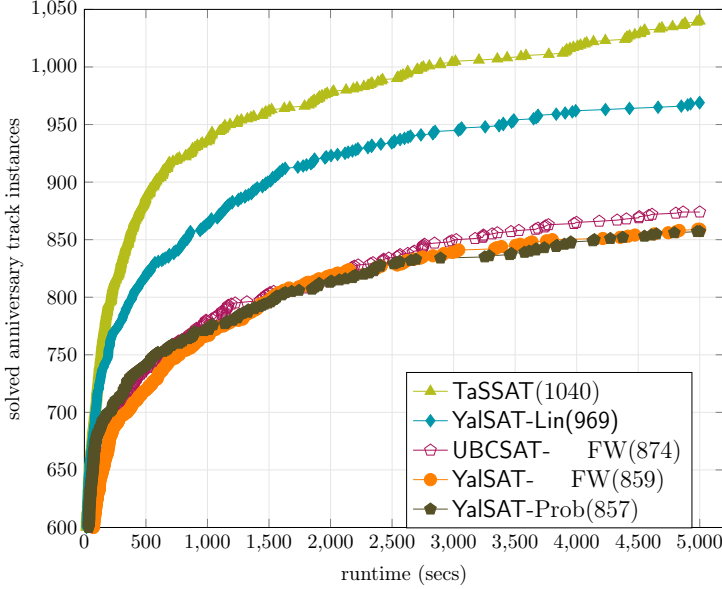


Fig. 2: Performance profiles for solver modifications on the **anni** benchmark set show that TaSSAT significantly outperforms the others. Since all solvers can quickly solve 600 instances, we start the y-axis at 600 to improve readability.

der Waerden number instances.⁴ For reproducibility, we set all randomization seeds to 0. For the **anni** instances, we ran TaSSAT and our baseline solvers in the StarExec Cluster [2] with a 5000-second timeout. For the van der Waeren instances, we ran the parallel version of TaSSAT with and without the ProbSAT-LiWeT option with a 48-hour timeout on the Bridges-2 cluster [8] with AMD EPYC 7742 CPUs (128 cores, 512GB RAM).

Figure 2 illustrates our results for the **anni** dataset. TaSSAT performed the best by solving 1040 problem instances, surpassing YalSAT-Lin, UBCSAT-DDFW, YalSAT-DDFW, and YalSAT-Prob with 969, 874, 859, and 857 solved instances, respectively. In particular, TaSSAT solved 71 more instances than YalSAT-Lin, the solver from our previous work, showing that our algorithmic changes are, in fact, improvements. The slight difference in solve counts between UBCSAT-DDFW and YalSAT-DDFW (874 vs. 859) can be attributed to random noise.

Notably, TaSSAT exclusively solved 12 instances that no 2022 SAT Competition solver could. However, YalSAT-Prob, YalSAT-Lin, UBCSAT-DDFW, and YalSAT-DDFW solved 73, 42, 40, and 38 **anni** instances, respectively, that TaSSAT could not.

We also present new lower bounds for van der Waerden numbers by running PaSSAT. The van der Waerden number $w(2; 3, t)$ is the smallest natural number n

⁴ Available at <https://github.com/solimul/vdw9>.

Table 1: Lower bounds for van der Waerden numbers $w(2; 3, t)$.

t	31	32	33	34	35	36	37	38	39
Ahmed et al. [3]	930	1006	1063	1143	1204	1257	1338	1378	1418
Our work	953	1011	1071	1145	1208	1260	1341	1380	1419

where for any partition of $\{1, \dots, n\}$ into P_0 and P_1 , either P_0 contains a 3-term arithmetic progression or P_1 contains a t -term arithmetic progression. In Table 1, we present in the top row previously-known lower bounds for $w(2; 3, t)$ for $31 \leq t \leq 39$.

The best lower bounds are obtained when PaSSAT leverages TaSSAT with the activation of the ProbSAT-LiWeT toggle and integrates YaISAT-style restarts. This configuration solves all 9 **vdw** benchmarks, pushing the lower bounds of these 9 numbers to values that are highlighted in the bottom row of Table 1. In contrast, using the default TaSSAT configuration, PaSSAT solves 7 **vdw** benchmarks, establishing same lower bounds for all the numbers shown in the bottom row of Table 1, except for $w(2; 3, 32)$ and $w(2; 3, 37)$. Hence, this version enhances the lower bounds for $w(2; 3, 32)$ and $w(2; 3, 37)$ to 1010 and 1340, respectively, just 1 short of their best-evaluated lower bounds. The performance of TaSSAT-Prob-LiWeT compared to TaSSAT-LiWeT is evident in their respective average PAR-2 scores, with values of 31,943 and 91,744.

Putting these results into perspective, Ahmed et al. [3] were unable to solve any of these **vdw** instances, despite employing 29 algorithms and extensive parallelization. Notably, the best result attained by Ahmed et al. using only SLS methods for $w(2; 3, 31)$ was 919. We improved this bound to 953. These results emphasize the unique algorithmic strengths of our solver.

In addition to improved solving, PaSSAT achieves significant memory reduction compared to our previous parallel solver [10]. Across the seven **vdw** benchmarks solved by both PaSSAT and the parallel solver, the average memory reduction is substantial, decreasing from 3.2 GB to 686.17 MB, a nearly 80% reduction. The reduction held even for the largest problem instance ($t = 39$), where the memory footprint decreased by nearly 80%, from 4.42 GB to 966 MB.

Code and Data Availability Statement

The code and data that support the contributions of this work are openly available in the ‘‘Artifact for TaSSAT: A Stochastic Local Search Solver for SAT’’ at <https://zenodo.org/records/10042124> [9]. The authors confirm that the data supporting the findings of this study are available within the article and the artifact.

References

1. SAT Competition 2022. <https://satcompetition.github.io/2022/downloads.html>, 2022.
2. Cesare Tinelli Aaron Stump, Geoff Sutcliffe. StarExec. <https://www.starexec.org/starexec/public/about.jsp>, 2013.
3. Tanbir Ahmed, Oliver Kullmann, and Hunter S. Snevily. On the van der Waerden numbers $w(2; 3, t)$. *Discrete Applied Mathematics*, 174:27–51, 2014.
4. Adrian Balint. *Engineering stochastic local search for the satisfiability problem*. PhD thesis, University of Ulm, 2014.
5. Adrian Balint, Armin Biere, Andreas Fröhlich, and Uwe Schöning. Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses. In *Proceedings of SAT-2014*, pages 302–316, 2014.
6. Armin Biere. YalSAT: Yet Another Local Search Solver. <http://fmv.jku.at/yalsat/>, 2010.
7. Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CADI-CAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING entering the SAT Competition. In *Proceedings of SAT Competition*, pages 50–53, 2020.
8. Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. Bridges-2: A platform for rapidly-evolving and data intensive research. In *Association for Computing Machinery, New York, NY, USA*, pages 1–4, 2021.
9. Md Solimul Chowdhury, Cayden Codel, and Marijn Heule. Artifact for TaSSAT: A stochastic local search solver for SAT.
10. Md Solimul Chowdhury, Cayden R. Codel, and Marijn J.H. Heule. A linear weight transfer rule for local search. In *NASA Formal Methods*, 2023.
11. Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.
12. Marijn J. H. Heule. Solving edge-matching problems with satisfiability solvers. In *SAT 2009 competitive events booklet*, pages 69–82, 2009.
13. Marijn J. H. Heule, Anthony Karahalios, and Willem-Jan van Hoeve. From cliques to colorings and back again. In *Proceedings of CP-2022*, pages 26:1–26:10, 2022.
14. Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. New ways to multiply 3×3 -matrices. *J. Symb. Comput.*, 104:899–916, 2019.
15. Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Commun. ACM*, 60(8):70–79, 2017.
16. Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for SAT. In *Proceedings of CP-2005*, Lecture Notes in Computer Science, pages 772–776, 2005.
17. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Proceedings of SAT-2009*, pages 244–257, 2009.

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

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





Speculative SAT Modulo SAT

V. K. Hari Govind¹✉, Isabel Garcia-Contreras¹, Sharon Shoham²,
and Arie Gurfinkel¹

¹ University of Waterloo, Waterloo, Canada
{hgvedira, igarciac, agurfink}@uwaterloo.ca

² Tel-Aviv University, Tel Aviv, Israel
sharon.shoham@cs.tau.ac.il

Abstract. State-of-the-art model-checking algorithms like IC3/PDR are based on uni-directional modular SAT solving for finding and/or blocking counterexamples. Modular SAT-solvers divide a SAT-query into multiple sub-queries, each solved by a separate SAT-solver (called a module), and propagate information (lemmas, proof obligations, blocked clauses, etc.) between modules. While modular solving is key to IC3/PDR, it is obviously not as effective as monolithic solving, especially when individual sub-queries are harder to solve than the combined query. This is partially addressed in SAT modulo SAT (SMS) by propagating unit literals back and forth between the modules and using information from one module to simplify the sub-query in another module as soon as possible (i.e., before the satisfiability of any sub-query is established). However, bi-directionality of SMS is limited because of the strict order between decisions and propagation – only one module is allowed to make decisions, until its sub-query is SAT. In this paper, we propose a generalization of SMS, called SPECSMS, that *speculates* decisions between modules. This makes it bi-directional – decisions are made in multiple modules, and learned clauses are exchanged in both directions. We further extend DRUP proofs and interpolation, these are useful in model checking, to SPECSMS. We have implemented SPECSMS in Z3 and empirically validate it on a series of benchmarks that are provably hard for SMS.

1 Introduction

IC3/PDR [3] is an efficient SAT-based Model Checking algorithm. Among many other innovations in IC3/PDR is the concept of a modular SAT-solver that divides a formula into multiple *frames* and each frame is solved by an individual SAT solver. The solvers communicate by exchanging proof obligations (i.e., satisfying assignments) and lemmas (i.e., learned clauses).

While modular reasoning in IC3/PDR is very efficient for a Model Checker, it is not as efficient as a classical monolithic SAT-solver. This is not surprising since modularity restricts the solver to colorable refutations [11], which are, in the worst case, exponentially bigger than unrestricted refutations. On the positive side, IC3/PDR's modular SAT-solving makes interpolation trivial, and enables

generalizations of proof obligations and inductive generalization of lemmas – both are key to the success of IC3/PDR.

This motivates the study of modular SAT-solving, initiated by SMS [1]. Our strategic vision is that our study will contribute to improvements in IC3/PDR. However, in this paper, we focus on modular SAT-solving in isolation.

In modular SAT-solving, multiple solvers interact to check satisfiability of a partitioned CNF formula, where each part of the formula is solved by one of the solvers. In this paper, for simplicity, we consider the case of two solvers $\langle S_s, S_m \rangle$ checking satisfiability of a formula pair $\langle \Phi_s, \Phi_m \rangle$. S_m is a *main* solver and S_s is a *secondary* solver. In the notation, the solvers are written right-to-left to align with IC3/PDR, where the main solver is used for frame 1 and the secondary solver is used for frame 0.

When viewed as a modular SAT-solver, IC3/PDR is uni-directional. First, S_m finds a satisfying assignment σ to Φ_m and only then, S_s extends σ to an assignment for Φ_s . Learned clauses, called *lemmas* in IC3/PDR, are only shared (or copied) from the secondary solver S_s to the main solver S_m .

SAT Modulo SAT (SMS) [1] is a modular SAT-solver that extends IC3/PDR by allowing inter-modular unit propagation and conflict analysis: whenever an interface literal is placed on a trail of any solver, it is shared with the other solver and both solvers run unit propagation, exchanging unit literals. This makes modular SAT-solving in SMS bi-directional as information flows in both directions between the solvers. Bi-directional reasoning can simplify proofs, but it significantly complicates conflict analysis. To manage conflict analysis, SMS does not allow the secondary solver S_s to make any decisions before the main solver S_m is able to find a complete assignment to its clauses. As a result, learned clauses are either local to each solver, or flow only from S_s to S_m , restricting the structure of refutations similarly to IC3/PDR.

Both IC3/PDR and SMS require S_m to find a complete satisfying assignment to Φ_m before the solving is continued in S_s . This is problematic since Φ_m might be hard to satisfy, causing them to get stuck in Φ_m , even if considering both formulas together quickly reveals the (un)satisfiability of $\langle \Phi_s, \Phi_m \rangle$.

In this paper, we introduce SPECSMS — a modular SAT-solver that employs a truly bi-directional reasoning. SPECSMS builds on SMS, while facilitating deeper communication between the modules by (1) allowing learnt clauses to flow in both directions, and (2) letting the two solvers interleave their decisions. The key challenge is in the adaptation of conflict analysis to properly handle the case of a conflict that depends on decisions over local variables of both solvers. Such a conflict cannot be explained to either one of the solvers using only interface clauses (i.e., clauses over interface variables). It may, therefore, require backtracking the search without learning any conflict clauses. To address this challenge, SPECSMS uses *speculation*, which tames decisions of the secondary solver that are interleaved with decisions of the main solver. If the secondary solver satisfies all of its clauses during speculation, a *validation* phase is employed, where the main solver attempts to extend the assignment to satisfy its unassigned clauses. If speculation leads to a conflict which depends on local deci-

sions of both solvers, *refinement* is employed to resolve the conflict. Refinement ensures progress even if no conflict clause can be learnt. With these ingredients, we show that SPECSMS is sound and complete (i.e., always terminates).

To certify SPECSMS’s result when it determines that a formula is unsatisfiable, we extract a *modular* clausal proof from its execution. To this end, we extend DRUP proofs [12] to account for modular reasoning, and devise a procedure for trimming modular proofs. Such proofs are applicable both to SPECSMS and to SMS. Finally, we propose an interpolation algorithm that extracts an interpolant [4] from a modular proof. Since clauses are propagated between the solvers in both directions, the extracted interpolants have the shape $\bigwedge_i (C_i \Rightarrow cls_i)$, where C_i are conjunctions of clauses and each cls_i is a clause.

Original SMS is implemented on top of MiniSAT. For this paper, we implemented both SMS and SPECSMS in Z3 [5], using the extendable SAT-solver interface of Z3. Thanks to its bi-directional reasoning, SPECSMS is able to efficiently solve both sat and unsat formulas that are provably hard for existing modular SAT-solvers, provided that speculation is performed at the right time. We describe a simple heuristic to decide when to speculate.

In summary, we make the following contributions: (i) the SPECSMS algorithm that leverages bi-directional modular reasoning (Sec. 3); (ii) modular DRUP proofs for SPECSMS (Sec. 4.1); (iii) proof-based interpolation algorithm; (iv) heuristics to guide speculation (Sec. 5); and (v) implementation and validation (Sec. 6).

2 Motivating examples

In this section, we discuss two examples in which both IC3/PDR-style unidirectional reasoning and SMS-style shallow bi-directional reasoning are ineffective. The examples illustrate why existing modular reasoning gets stuck. To better convey our intuition, we present our problems at word level using bit-vector variables directly, without explicitly converting them to propositional variables.

Example 1. Consider the following modular sat query: $\langle \varphi_{in}, \varphi_{\text{SHA-1}} \rangle$, where $\varphi_{in} \triangleq (in = in_1) \vee (in = in_2)$, in is a 512-bit vector, in_1, in_2 are 512-bit values, $\varphi_{\text{SHA-1}} \triangleq (\text{SHA-1}_{\text{circ}}(in) = \text{SHA-1}_{in_1})$, $\text{SHA-1}_{\text{circ}}(in)$ is a circuit that computes SHA-1 of in , and SHA-1_{in_1} is the 20 byte SHA-1 message digest of in_1 .

Checking the satisfiability of $\varphi_{in} \wedge \varphi_{\text{SHA-1}}$ is easy because it contains both the output and the input of the SHA-1 circuit. However, existing modular SAT-solvers attempt to solve the problem starting by finding a complete satisfying assignment to $\varphi_{\text{SHA-1}}$. This is essentially the problem of inverting the SHA-1 function, which is known to be very hard for a SAT-solver. The improvements in SMS allow unit propagation between the two modules. However, this does not help since there are no unit clauses in φ_{in} .

On the other hand, SPECSMS proceeds as follows: (1) when checking satisfiability of $\varphi_{\text{SHA-1}}$, it decides to speculate, (2) it starts checking satisfiability of φ_{in} , branches on variables in , finds an assignment σ to in and unit propagates σ to

$\varphi_{\text{SHA-1}}$, (3) if there is a conflict in $\varphi_{\text{SHA-1}}$, it learns the conflict clause $in \neq in_2$, and (4) it terminates with a satisfying assignment $in = in_1$. Speculation in step (1) is what differentiates SPECSMS from IC3/PDR and SMS. The specifics of when exactly SPECSMS speculates is guided by a heuristic that is explained in Sec. 5.

Example 2. Speculation is desirable for unsatisfiable formulas as well. Consider the modular sat query $\langle \varphi_+, \varphi_- \rangle$, where $\varphi_+ \triangleq (a < 0 \Rightarrow x) \wedge (a \geq 0 \Rightarrow x) \wedge PHP_{32}^1$ and $\varphi_- \triangleq (b < 0 \Rightarrow \neg x) \wedge (b \geq 0 \Rightarrow \neg x) \wedge PHP_{32}^2$. Here, a and b are 32-wide bitvectors and local to the respective modules. PHP_{32} encodes the problem of fitting 32 pigeons into 31 holes and PHP_{32}^1 and PHP_{32}^2 denote a partitioning of PHP_{32} into 2 problems such that both formulas contain all variables. The modular problem $\langle \varphi_+, \varphi_- \rangle$ is unsatisfiable, x and PHP_{32}^1 being two possible interpolants. IC3/PDR and SMS only find the second interpolant. This is because, all satisfying assignments to φ_- immediately produce a conflict in PHP_{32}^1 part of φ_+ , without having to make any decisions. However, learning an interpolant containing x requires searching (i.e., deciding) in both φ_+ and φ_- . SPECSMS solves this problem by speculating right after deciding on all b variables. During speculation, the secondary solver hits a conflict on x once it tries to find an assignment to a variables. Note here that speculating after finding assignments to b variables and before finding an assignment to PHP_{32}^2 is crucial for SPECSMS to find the small interpolant.

These examples highlight the need to speculate while doing modular reasoning. Even though speculation by itself is quite powerful, to make SPECSMS effective in practice, we need good heuristics to decide when to enter speculation. We discuss some simple heuristics in Sec. 5.

3 Speculative SAT Modulo SAT

This section presents SPECSMS — a modular bi-directional SAT algorithm. For simplicity, we restrict our attention to the case of two modules. However, the algorithm easily generalizes to any sequence of modules.

3.1 Sat Modulo Sat

We assume that the reader has some familiarity with internals of a MiniSAT-like SAT solver [6] and with SMS [1]. We give a brief background on SMS, highlighting some of the key aspects. SMS decides satisfiability of a partitioned CNF formula $\langle \Phi_s, \Phi_m \rangle$ with a set of shared interface variables I . It uses two modules $\langle S_s, S_m \rangle$, where S_m is a *main* module used to solve Φ_m , and S_s is a *secondary* module to solve Φ_s . Each module is a SAT solver (with a slightly extended interface, as described in this section). We refer to them as *modules* or *solvers*, interchangeably. Each solver has its own clause database (initialized with Φ_i for $i \in \{m, s\}$), and a trail of literals, just as a regular SAT solver. The solvers keep their decision levels in sync. Whenever a decision is made in one

solver, the decision level of the other solver is incremented as well (adding a *null* literal to its trail if necessary). Whenever one solver back-jumps to level i , the other solver back-jumps to level i as well. Assignments to interface variables are shared between the solvers: whenever such a literal is added to the trail of one solver (either as a decision or due to propagation), it is also added to the trail of the other solver. SMS requires that S_s does not make any decisions, until S_m finds a satisfying assignment to its clauses.

Inter-modular propagation and conflict analysis The two key features of SMS are inter-modular unit propagation (called PROPAGATEALL in [1]) and the corresponding inter-modular conflict analysis. In PROPAGATEALL, whenever an interface literal is added to the trail of one solver, it is added to the trail of the other, and both solvers run unit propagation. Whenever a unit literal ℓ is copied from the trail of one solver to the other, the **reason** for ℓ in the destination solver is marked using a marker `ext`. This indicates that the justification for the unit is external to the destination solver³. Propagation continues until either there are no more units to propagate or one of the solvers hits a conflict.

Conflict analysis in SMS is extended to account for units with no reason clauses. If such a literal ℓ is used in conflict analysis, its reason is obtained by using `AnalyzeFinal(ℓ)` on the other solver to compute a clause ($s \Rightarrow \ell$) over the interface literals. This clause is copied to the requesting solver and is used as the missing reason. Multiple such clauses can be copied (or learned) during analysis of a single conflict clause – one clause for each literal in the conflict that is assigned by the other solver.

In SMS, it is crucial that `AnalyzeFinal(ℓ)` always succeeds to generate a reason clause over the interface variables. This is ensured by only calling `AnalyzeFinal(ℓ)` in the S_s solver on literals that were added to the trail when S_s was not yet making decisions. This can happen in one of two scenarios: either S_m hits a conflict due to literals propagated from S_s , in which case `AnalyzeFinal` is invoked in S_s on each literal marked `ext` in S_m that is involved in the conflict resolution to obtain its **reason**; or S_s hits a conflict during unit propagation, in which case it invokes `AnalyzeFinal` to obtain a conflict clause over the interface variables that blocks the partial assignment of S_m . In both cases, new reason clauses are always copied from S_s to S_m . We refer the reader to [1] for the pseudo-code of the above inter-modular procedures for details.

3.2 Speculative Sat Modulo Sat

SPECSMS extends SMS [1] by a combination of *speculation*, *refinement*, and *validation*. During the search in the main solver S_m , SPECSMS non-deterministically *speculates* by allowing the secondary solver S_s to extend the current partial assignment of Φ_m to a satisfying assignment of Φ_s . If S_s is unsuccessful (i.e., hits a conflict), and the conflict depends on a combination of a *local* decision of S_m

³ This is similar to theory propagation in SMT solvers.

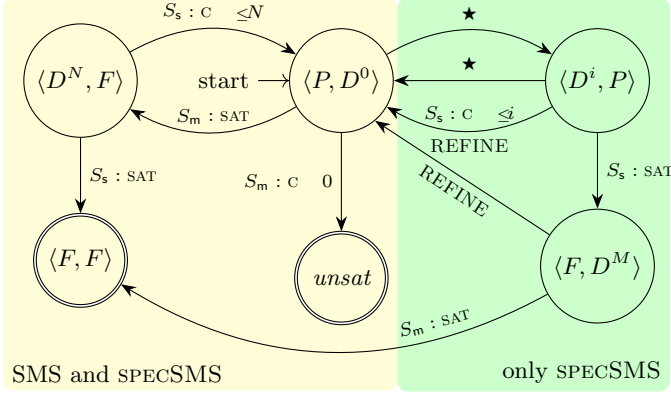


Fig. 1: State transitions of SPECSMS. A state $\langle P, D^0 \rangle$ means that the secondary solver S_s is in propagate mode and the main solver S_m is in decide mode. Each edge is guarded with a condition. The condition $S_m : SAT$ means that S_m found a full satisfying assignment to Φ_m . The condition $S_m : c @ \leq j$ means that S_m hit a conflict at a decision level below j . The four states in yellow corresponds to SMS; two states in green are unique to SPECSMS.

with some decision of S_s , then the search reverts to S_m and its partial assignment is *refined* by forcing S_m to decide on an interface literal from the conflict. On the other hand, if S_s is successful, solving switches to the main solver S_m that *validates* the current partial assignment by extending it to all of its clauses. This either succeeds (meaning, $\langle \Phi_s, \Phi_m \rangle$ is sat), or fails and another *refinement* is initiated. Note that the two sub-cases where S_s is unsuccessful but the reason for the conflict is either local to S_s or local to S_m are handled as in SMS.

Search modes SPECSMS controls the behavior of the solvers and their interaction through *search modes*. Each solver can be in one of the following search modes: Decide, Propagate, and Finished. In Decide, written D^i , the solver treats all decisions below level i as assumptions and is allowed to both make decisions and do unit propagation. In Propagate, written P , the solver makes no decisions, but does unit propagation whenever new literals are added to its trail. In Finished, written F , the clause database of the solver is satisfied; the solver neither makes decisions nor propagates unit literals.

The pair of search modes of both modules is called the *state* of SPECSMS, where we add a unique state called *unsat* for the case when the combination of the modules is known to be unsatisfiable. The possible states and transitions of SPECSMS are shown in Fig. 1. States *unsat* and $\langle F, F \rangle$ are two final states, corresponding to unsat and sat, respectively. In all other states, exactly one of the solvers is in a state D^i . We refer to this solver as *active*. The part of the transition system highlighted in yellow correspond to SMS, and the green part includes the states and transitions that are unique to SPECSMS.

Normal execution with bi-directional propagation SPECSMS starts in the state $\langle P, D^0 \rangle$, with the main solver being active. In this state, it can proceed like SMS by staying in the yellow region of Fig. 1. We call this *normal execution with bi-directional propagation*, since (only) unit propagation goes between solvers.

Speculation What sets SPECSMS apart is speculation: at any non-deterministically chosen decision level i , SPECSMS can pause deciding on the main solver and activate the secondary solver (i.e., transition to state $\langle D^i, P \rangle$). During speculation, only the secondary solver makes decisions. Since the main solver does not have a full satisfying assignment to its clauses, the secondary solver propagates assignments to the main solver and vice-versa.

Speculation terminates when the secondary solver S_s either: (1) hits a conflict that cannot be resolved by inter-modular conflict analysis; (2) hits a conflict below decision level i ; or (3) finds a satisfying assignment to Φ_s .

Case (1) is most interesting, and is what makes SPECSMS differ from SMS. Note that a conflict clause is not resolved by inter-modular conflict analysis only if it depends on an external literal on the trail of S_s that cannot be explained by an interface clause from S_m . This is possible when both S_m and S_s have partial assignments during speculation. So the conflict might depend on the *local* decisions of S_m . This cannot be communicated to S_s using only interface variables.

Refinement In SPECSMS, this is handled by modifying the REASON method in the solvers to fail (i.e., return ext) whenever **AnalyzeFinal** returns a non-interface clause. Additionally, the literal on which **AnalyzeFinal** failed is recorded in a global variable *refineLit*. This is shown in Alg. 1. The inter-modular conflict analysis is modified to exit early whenever REASON fails to produce a justification. At this point, SPECSMS exits speculation, returns to the initial state $\langle P, D^0 \rangle$, both solvers back-jump to decision level i at which speculation was initiated, and S_m is forced to decide on *refineLit*.

We call this transition a *refinement* because the partial assignment of the main solver S_m (which we view as an *abstraction*) is updated (a.k.a., refined) based on the information that was not available to it (namely, a conflict with a set of decisions in the secondary solver S_s). Since *refineLit* was not decided on in S_m prior to speculation, deciding on it is a new decision that ensures progress in S_m . The next speculation is possible only under strictly more decisions in S_m than before, or when S_m back-jumps and flips an earlier decision.

We illustrate the refinement process on a simple example:

Example 3. Consider the query $\langle \Phi_s, \Phi_m \rangle$ with:

$\Phi_s(i, j, k, z):$		$\Phi_m(a, i, j, k):$
$\bar{z} \vee \bar{i}$		$\bar{a} \vee i \vee \bar{j}$
$i \vee j \vee \bar{k}$		$j \vee k$

First, S_m decides a (at level 1), which causes no propagations. Then, SPECSMS

enters speculative mode, transitions to $\langle D^1, P \rangle$ and starts making decisions in S_s . S_s decides z and calls PROPAGATEALL. Afterwards, the trails for S_m and S_s are as follows:

S_m	$a @ 1$	$null @ 2$	$\bar{i} \text{ (ext)}$	$\bar{j} (1)$	$k (2)$
S_s	$null @ 1$	$z @ 2$	$\bar{i} (3)$	$\bar{j} \text{ (ext)}$	$k \text{ (ext)}$

where $x @ i$ denotes that literal x is decided at level i , and $x (r)$ denotes that literal x is propagated using a reason clause r , or due to the other solver (if $r = \text{ext}$). A conflict is hit in S_s in clause (4). Inter-modular conflict analysis begins. S_s first asks for the reason for k , which is clause (2) in S_m . This clause is copied to S_s . Note that unlike SMS, clauses can move from S_m to S_s . The new conflict to be analyzed is $(i \vee j \vee \bar{j})$. Now the reason for \bar{j} is asked of S_m . In this case, S_m cannot produce a clause over shared variables to justify \bar{j} , so conflict analysis fails with $\text{refineLit} = j$. This causes SPECSMS to exit speculation mode and move to state $\langle P, D^0 \rangle$ and S_m must decide variable j before speculating again. In this case either decision on j results in $\langle \Phi_s, \Phi_m \rangle$ being sat. \square

In addition to refining when conflict analysis fails, SPECSMS also has the ability to refine non-deterministically. That is, at any point during speculation, S_s can decide to stop speculation, back-jump to the decision level from which it started speculation, and choose any interface literal as refineLit .

Case (2) is similar to what happens in SMS when a conflict is detected in S_s . The reason for the conflict is below level i which is below the level of any decision of S_s . Since decision levels below i are treated as assumptions in S_s , calling `AnalyzeFinal` in S_s returns an interface clause c that blocks the current assignment in S_m . The clause c is added to S_m . The solvers back-jump to the smallest decision level j that makes c an asserting clause in S_m . Finally, SPECSMS moves to $\langle P, D^0 \rangle$.

Validation Case (3), like Case (1), is unique to SPECSMS. While all clauses of S_s are satisfied, the current assignment might not satisfy all clauses of S_m . Thus, SPECSMS enters *validation* by switching to the configuration $\langle F, D^M \rangle$, where M is the current decision level. Thus, S_m becomes active and starts deciding and propagating. This continues, until one of two things happen: (3a) S_m extends the assignment to satisfy all of its clauses, or (3b) a conflict that cannot be resolved with inter-modular conflict analysis is found. In the case (3a), SPECSMS transitions to $\langle F, F \rangle$ and declares that $\langle \Phi_m, \Phi_s \rangle$ is sat. The case (3b) is handled exactly the same as Case (1) – the literal on the trail without a reason is stored in refineLit , SPECSMS moves to $\langle P, D^0 \rangle$, backjumps to the level in which speculation was started, and S_m is forced to decide on refineLit .

Theorem 1. *SPECSMS terminates. If it reaches the state $\langle F, F \rangle$, then $\Phi_s \wedge \Phi_m$ is satisfiable and the join of the trails of $\langle S_s, S_m \rangle$ is a satisfying assignment. If it reaches the state *unsat*, $\Phi_s \wedge \Phi_m$ is unsatisfiable.*

Algorithm 1 The REASON method in modular SAT solvers inside SPECSMS

```

1: function REASON(lit)
2:   if reason[lit] = ext then
3:     c ← other.AnalyzeFinal(lit)
4:     if  $\exists v \in c :: v \notin I$  then
5:       refineLit ← lit
6:       return ext
7:   ADDCLAUSE(c)
8:   reason[lit] ← c
9:   return reason[lit]

```

4 Validation and interpolation

In this section, we augment SPECSMS with an interpolation procedure. To this end, we first introduce modular DRUP proofs, which are generated from SPECSMS in a natural way. We then present an algorithm for extracting an interpolant from a modular trimmed DRUP proof in the spirit of [11].

4.1 DRUP proofs for modular SAT

Modular DRUP proofs – a form of clausal proofs [9] – extend (monolithic) DRUP proofs [12]. A DRUP proof [12] is a sequences of steps, where each step either asserts a clause, deletes a clause, or adds a new Reverse Unit Propagation (RUP) clause. Given a set of clauses Γ , a clause *cls* is an RUP for Γ , written $\Gamma \vdash_{UP} cls$, if *cls* follows from Γ by unit propagation [8]. For a DRUP proof π , let ASSERTED(π) denote all clauses of the asserted commands in π , then π shows that all RUP clauses of π follow from ASSERTED(π). If π contains a \perp clause, then π certifies ASSERTED(π) is unsat.

A Modular DRUP proof is a sequence of clause addition and deletion steps, annotated with indices *idx* (m or s). Intuitively, steps with the same index must be validated together (within the same module *idx*), and steps with different indices may be checked independently. The steps are:

1. (asserted, *idx*, *cls*) denotes that *cls* is asserted in *idx*,
2. (rup, *idx*, *cls*) denotes adding RUP clause *cls* to *idx*,
3. (cp(*src*), *dst*, *cls*) denotes copying a clause *cls* from *src* to *dst*, and
4. (del, *idx*, *cls*) denotes removing clause *cls* from *idx*.

We denote the prefix of length k of a sequence of steps π by π^k . Given a sequence of steps π and a formula index *idx*, we use *act_clauses*(π , *idx*) to denote the set of active clauses with index *idx*. Formally,

$$\begin{aligned}
& \{cls \mid \exists c_j \in \pi \cdot \\
& \quad (c_j = (t, idx, cls) \wedge (t = \text{asserted} \vee t = \text{rup} \vee t = \text{cp}(_))) \\
& \quad \wedge \neg \exists c_k \in \pi \cdot k > j \wedge c_k = (\text{del}, idx, cls)\}
\end{aligned}$$

seq step	to	clause
1 asserted	m	$\neg s_1 \Rightarrow lb_1$
2 asserted	m	$\neg s_1 \Rightarrow \neg lb_1$
3 asserted	s	$(s_1 \wedge la_1) \Rightarrow s_2$
4 asserted	s	$(s_1 \wedge \neg la_1) \Rightarrow s_2$
5 asserted	m	$(s_2 \wedge lb_2) \Rightarrow s_3$
6 asserted	m	$(s_2 \wedge \neg lb_2) \Rightarrow s_3$
7 asserted	s	$(s_3 \wedge la_2) \Rightarrow s_4$
8 asserted	s	$(s_3 \wedge \neg la_2) \Rightarrow s_4$
9 asserted	m	$s_4 \Rightarrow lb_3$
10 asserted	m	$s_4 \Rightarrow \neg lb_3$
11 rup	m	s_1
12 rup	m	$\neg s_4$
13 rup	m	$s_2 \Rightarrow s_3$
14 cp(m)	s	$s_2 \Rightarrow s_3$
15 rup	s	$s_3 \Rightarrow s_4$
16 rup	s	$s_1 \Rightarrow s_4$
17 cp(s)	m	$s_1 \Rightarrow s_4$
18 rup	m	\perp

Fig. 2: An example of a modular DRUP proof. Clauses are written in human-readable form as implications, instead of in the DIMACS format.

A sequence of steps $\pi = c_1, \dots, c_n$ is a *valid modular DRUP proof* iff for each $c_i \in \pi$:

1. if $c_i = (\text{rup}, idx, cls)$ then $\text{act_clauses}(\pi^i, idx) \vdash_{UP} cls$,
2. if $c_i = (\text{cp}(idx), -, cls)$ then $\text{act_clauses}(\pi^i, idx) \vdash_{UP} cls$, and
3. $c_{|\pi|}$ is either (rup, m, \perp) or $(\text{cp}(s), m, \perp)$.

Let $\text{ASSERTED}(\pi, idx)$ be the set of all asserted clauses in π with index idx .

Theorem 2. *If π is a valid modular DRUP proof, then $\text{ASSERTED}(\pi, s) \wedge \text{ASSERTED}(\pi, m)$ is unsatisfiable.*

Modular DRUP proofs may be validated with either one or two solvers. To validate with one solver we convert the modular proof into a monolithic one (i.e., where the steps are asserted, rup, and del). Let MODDRUP2DRUP be a procedure that given a modular DRUP proof π , returns a DRUP proof π' that is obtained from π by (a) removing idx from all the steps; (b) removing all cp steps; (c) removing all del steps. Note that del steps are removed for simplicity, otherwise it is necessary to account for deletion of copied and non-copied clauses separately.

Lemma 1. *If π is a valid modular DRUP proof then $\pi' = \text{MODDRUP2DRUP}(\pi)$ is a valid DRUP proof.*

Modular validation is done with two monolithic solvers working in lock step: (asserted, cls, idx) steps are added to the idx solver; (rup, idx, cls) steps are validated locally in solver idx using all active clauses (asserted, copied, and rup);

and for $(cp(src), dst, cls)$ steps, cls is added to dst but not validated in it, and cls is checked to exist in the src solver.

From now on, we consider only valid proofs. We say that a (valid) modular DRUP proof π is a proof of unsatisfiability of $\Phi_s \wedge \Phi_m$ if $\text{ASSERTED}(\pi, s) \subseteq \Phi_s$ and $\text{ASSERTED}(\pi, m) \subseteq \Phi_m$ (inclusion here refers to the sets of clauses).

SPECSMS produces modular DRUP proofs by logging the clauses that are learnt, deleted, and copied between solvers. Note that in SMS clauses may only be copied from S_s to S_m , but in SPECSMS they might be copied in both directions.

Theorem 3. *Let Φ_s and Φ_m be two Boolean formulas s.t. $\Phi_s \wedge \Phi_m \models \perp$. SPECSMS produces a valid modular DRUP proof for unsatisfiability of $\Phi_s \wedge \Phi_m$.*

Algorithm 2 Trimming a modular DRUP proof

Input: Solver instances S_s, S_m with the empty clause on the trail, and a modular clausal proof $\pi = c_1, \dots, c_n$.

Output: A proof π' s.t. all steps are core.

```

1:  $\pi' = \emptyset$ 
2:  $M_s, M_m \leftarrow \{\perp\}, \emptyset$   $\triangleright$  Relevant clauses
3: for  $i = n$  to 0 do
4:   match  $c_i$  with  $(type, idx, cls)$ 
5:   if  $cls \notin M_{idx}$  then continue
6:   if  $type = \text{del}$  then
7:      $S_{idx}.\text{Revoke}(cls)$ 
8:     continue
9:    $\pi'.\text{append}(c_i)$ 
10:  if  $type = \text{rup}$  then
11:     $S_{idx}.\text{CHK\_RUP}(cls, M_{idx})$ 
12:  else if  $type = \text{cp}(src)$  then
13:     $S_{idx}.\text{Delete}(cls)$ 
14:     $M_{src}.\text{add}(cls)$ 
15:  $\pi'.\text{reverse}()$ 
16: function  $\text{SOLVER}::\text{CHK\_RUP}(cls, M)$ 
17:   if  $\text{IsOnTrail}(cls)$  then
18:      $\text{UndoTrail}(cls)$ 
19:    $\text{Delete}(cls)$ 
20:    $\text{SaveTrail}()$ 
21:    $\text{Enqueue}(\neg cls)$ 
22:    $r \leftarrow \text{Propagate}()$ 
23:    $\text{ConflictAnalysis}(r, M)$   $\triangleright$  Updates  $M$  with
      conflict clauses
24:    $\text{RestoreTrail}()$ 
```

Algorithm 3 Interpolating a modular DRUP proof.

Input: Propositional formulas $\langle \Phi_0, \Phi_1 \rangle$

Input: A modular trimmed DRUP proof $\pi = c_1, \dots, c_n$ of unsatisfiability of $\Phi_0 \wedge \Phi_1$

Output: An interpolant itp s.t. $\Phi_0 \Rightarrow itp$ and $itp \wedge \Phi_1 \models \perp$

```

1:  $S_s, S_m \leftarrow \text{SAT\_SOLVER}()$ 
2:  $itp \leftarrow \top$ 
3: for  $i = 0$  to  $n$  do
4:   match  $c_i$ 
5:   with  $(\text{asserted}, s, cls)$ :
6:      $\text{sup}(cls) \leftarrow \top$ 
7:   with  $(\text{cp}(m), s, cls)$ :
8:      $\text{sup}(cls) \leftarrow cls$ 
9:   with  $(\text{rup}, s, cls)$ :
10:     $M \leftarrow \emptyset$ 
11:     $S_s.\text{CHK\_RUP}(cls, M)$ 
12:     $\text{sup}(cls) \leftarrow \{\text{sup}(c) \mid c \in M\}$ 
13:   with  $(\text{cp}(s), m, cls)$ :
14:     $itp \leftarrow itp \wedge (\text{sup}(cls) \Rightarrow cls)$ 
15:    $S_{c_i.idx}.\text{add}(cls)$ 
```

Trimming modular DRUP proofs. A step in a modular DRUP proof π is *core* if removing it invalidates π . Under this definition, del steps are never core since

removing them does not affect validation. Alg. 2 shows an algorithm to trim modular DRUP proofs based on backward validation. The input are two modular solvers S_m and S_s in a final conflicting state, and a valid modular DRUP proof $\pi = c_1, \dots, c_n$. The output is a trimmed proof π' s.t. all steps of π' are core.

We assume that the reader is familiar with MiniSAT [6] and use the following solver methods: **Propagate**, exhaustively applies unit propagation (UP) rule by resolving all unit clauses; **ConflictAnalysis** analyzes the most recent conflict and marks which clauses are involved in the conflict; **IsOnTrail** checks whether a clause is an antecedent of a literal on the trail; **Enqueue** enqueues one or more literals on the trail; **IsDeleted**, **Delete**, **Revive** check whether a clause is deleted, delete a clause, and add a previously deleted clause, respectively; **SaveTrail**, **RestoreTrail** save and restore the state of the trail.

Alg. 2 processes the steps of the proof backwards, rolling back the states of the solvers. M_{idx} marks which clauses were relevant to derive clauses in the current suffix of the proof. While the proof is constructed through inter-modular reasoning, the trimming algorithm processes each of the steps in the proof completely locally. During the backward construction of the trimmed proof, steps that include unmarked clauses are ignored (and, in particular, not added to the proof). For each (relevant) rup step, function **CHK_RUP**, using **ConflictAnalysis**, adds clauses to M . **del** steps are never added to the trimmed proof, but the clause is revived from the solver. For **cp** steps, if the clause was marked, it is marked as used for the solver it was copied from and the step is added to the proof. Finally, asserted clauses that were marked are added to the trimmed proof. Note that, as in [11], proofs may be trimmed in different ways, depending on the strategy for **ConflictAnalysis**.

The following theorem states that trimming preserves validity of the proof:

Theorem 4. *Let Φ_s and Φ_m be two formulas such that $\Phi_s \wedge \Phi_m \models \perp$. If π is a modular DRUP proof produced by solvers S_s and Φ_m for $\Phi_s \wedge \Phi_m$, then a trimmed proof π' by Alg. 2 is also a valid modular DRUP proof for $\Phi_s \wedge \Phi_m$.*

Fig. 2 shows a trimmed proof after SPECSMS is executed on $\langle \psi_0, \psi_1 \rangle$ such that $\psi_0 \triangleq ((s_1 \wedge la_1) \Rightarrow s_2) \wedge ((s_1 \wedge \neg la_1) \Rightarrow s_2) \wedge ((s_3 \wedge la_2) \Rightarrow s_4) \wedge ((s_3 \wedge \neg la_2) \Rightarrow s_4)$ and $\psi_1 \triangleq (\neg s_1 \Rightarrow lb_1) \wedge (\neg s_1 \Rightarrow \neg lb_1) \wedge ((s_2 \wedge lb_2) \Rightarrow s_3) \wedge ((s_2 \wedge \neg lb_2) \Rightarrow s_3) \wedge (s_4 \Rightarrow lb_3) \wedge (s_4 \Rightarrow \neg lb_3)$.

4.2 Interpolation

Given a modular DRUP proof π of unsatisfiability of $\Phi_s \wedge \Phi_m$, we give an algorithm to compute an interpolant of $\Phi_s \wedge \Phi_m$. For simplicity of the presentation, we assume that π has no deletion steps; this is the case in trimmed proofs, but we can also adapt the interpolation algorithm to handle deletions by keeping track of active clauses.

Our interpolation algorithm relies only on the clauses copied between the modules. Notice that whenever a clause is copied from module i to module j , it is implied by all the clauses in Φ_i together with all the clauses that have been copied from module j . We refer to clauses copied from S_m to S_s as *backward*

clauses and clauses copied from S_s to S_m as *forward* clauses. The conjunction of forward clauses is unsatisfiable with S_m . This is because, in the last step of π , \perp is added to S_m , either through rup or by $\text{cp } \perp$ from S_s . Since all the clauses in module m are implied by Φ_m together with forward clauses, this means that the conjunction of forward clauses is unsatisfiable with Φ_m . In addition, all forward clauses were learned in module s , with support from backward clauses. This means that every forward clause is implied by Φ_s together with the subset of the backward clauses used to derive it. Intuitively, we should therefore be able to learn an interpolant with the structure: backward clauses imply forward clauses.

Alg. 3 describes our interpolation algorithm. It traverses a modular DRUP proof forward. For each clause cls learned in module s , the algorithm collects the set of backward clauses used to learn cls . This is stored in the sup datastructure — a mapping from clauses to sets of clauses. Finally, when a forward clause c is copied, it adds $\text{sup}(c) \Rightarrow c$ to the interpolant.

Example 4. We illustrate our algorithm using the modular DRUP proof from Fig. 2. On the first cp step ($\text{cp}(m), s, s_2 \Rightarrow s_3$), the algorithm assigns the sup for clause $s_2 \Rightarrow s_3$ as itself (line 8). The first clause learnt in module s , ($\text{rup}, s, s_3 \Rightarrow s_4$), is derived from just the clauses in module s and no backward clauses. Therefore, after RUP, our algorithm sets $\text{sup}(s_3 \Rightarrow s_4)$ to \top (line 12). The second clause learnt in module s , $s_1 \Rightarrow s_4$, is derived from module s with the support of the backward clause $s_2 \Rightarrow s_3$. Therefore, $\text{sup}(s_1 \Rightarrow s_4) = \{s_2 \Rightarrow s_3\}$. When this clause is copied forward to module 1, the algorithm updates the interpolant to be $(s_2 \Rightarrow s_3) \Rightarrow (s_1 \Rightarrow s_4)$. \square

Next, we formalize the correctness of the algorithm. Let $L_B(\pi) = \{cls \mid (\text{cp}(m), s, cls) \in \pi\}$ be the set of clauses copied from module m to s and $L_F(\pi) = \{cls \mid (\text{cp}(s), m, cls) \in \pi\}$ be clauses copied from module s to m . From the validity of modular DRUP proofs, we have that:

Lemma 2. *For any step $c_i = (\text{cp}(s), m, cls) \in \pi$, $(L_B(\pi^i) \wedge \Phi_s) \Rightarrow cls$ and for any step $c_j = (\text{cp}(m), s, cls) \in \pi$, $(L_F(\pi^j) \wedge \Phi_m) \Rightarrow cls$.*

For any clause cls copied from one module to the other, we use the shorthand $\sharp(cls)$ to refer to the position of the copy command in the proof π . That is, $\sharp(cls)$ is the smallest k such that $c_k = (\text{cp}(i), j, cls) \in \pi$. The following is an invariant in a valid modular DRUP proof:

Lemma 3.

$$\forall cls \in L_F(\pi) \cdot (\Phi_m \wedge (L_F(\pi^{\sharp(cls)})) \Rightarrow L_B(\pi^{\sharp(cls)}))$$

These properties ensure that adding $L_B(\pi^{\sharp(cls)}) \Rightarrow cls$ for every forward clause cls results in an interpolant. Alg. 3 adds $(\text{sup}(cls) \Rightarrow cls)$ as an optimization. Correctness is preserved since $\text{sup}(cls)$ is a subset of $L_B(\pi^{\sharp(cls)})$ that together with Φ_s suffices to derive cls (formally, $\text{sup}(cls) \wedge \Phi_s \vdash_{UP} cls$).

Theorem 5. *Given a modular DRUP proof π for $\Phi_s \wedge \Phi_m$, $\text{itp} \triangleq \{\text{sup}(c) \Rightarrow c \mid c \in L_F(\pi)\}$ is an interpolant for $\langle \Phi_s, \Phi_m \rangle$.*

Proof. Since all copy steps are over interface variables, the interpolant is also over interface variables. By Lemma 2 (and the soundness of sup optimization), $\Phi_s \Rightarrow itp$. Next, we prove that $(\Phi_m \wedge itp) \Rightarrow \perp$. From Lemma 3, we have that for all $c \in L_F(\pi)$, $(\Phi_m \wedge L_F(\pi^{\sharp(c)})) \Rightarrow \text{sup}(c)$. Therefore, $(\Phi_m \wedge L_F(\pi^{\sharp(c)}) \wedge (\text{sup}(c) \Rightarrow c)) \Rightarrow c$

It is much simpler to extract interpolants from modular DRUP proofs than from arbitrary DRUP proofs. This is not surprising since the interpolants capture exactly the information that is exchanged between solvers. The interpolants are not in CNF, but can be converted to CNF after extraction.

5 Heuristics for guiding specSMS

Theoretically, speculation makes SPECSMS more powerful than SMS and IC3/PDR. However, in practice, deciding when to enter speculation has a major impact on the performance of SPECSMS. If the speculation is too greedy, SPECSMS performs poorly on examples where the main module is easy to solve. Similarly, if the speculation is too lazy, SPECSMS performs poorly on problems in which any solution to the secondary module makes the main module easy to solve. We illustrate this trade-off using an example.

Example 5. Consider a modular query: $\langle \gamma_{in}(\ell, x, in), \gamma_{\text{SHA-1}}(in, x, out) \rangle$, where x is an 512-bit vector, ℓ is a 160-bit vector, $chks_i$ are 512-bit vector, and the remaining variables are the same as in ψ_{in} and $\psi_{\text{SHA-1}}$, and

$$\begin{aligned} \gamma_{in} &\triangleq \text{SHA-1}_{\text{circ}}(x, \ell) \wedge \\ &\quad ((\ell = chks_0 \wedge in = msg_0) \vee (\ell = chks_1 \wedge in = msg_1) \vee \\ &\quad (\ell = chks_2 \wedge in = msg_2) \vee (\ell = chks_3 \wedge in = msg_3)) \\ \gamma_{\text{SHA-1}} &\triangleq (x = 1 \vee x = 4) \wedge \text{SHA-1}_{\text{circ}}(in, out) \wedge out = shaVal \end{aligned}$$

This is an example where bi-directional search is necessary to efficiently solve the query. If deciding only on $\gamma_{\text{SHA-1}}$, we encounter the hard problem of inverting $\text{SHA-1}_{\text{circ}}$, if deciding in γ_{in} , we encounter the same problem, since an assignment for x needs to be found, based on the four values for ℓ . Therefore, neither immediate nor late speculation makes SPECSMS efficient on the problem. The ideal strategy here is to speculate after an assignment to x , to simplify γ_{in} . \square

Ideally, we would like to speculate when the current modular query is too hard for the solver. As a proxy for hardness, we measure the number of conflicts the SAT solver hits. We first speculate when the main solver hits a predetermined number of conflicts. We then exponentially widen the number of conflicts between speculations. Exiting from speculation is just as important as entering speculation: the secondary solver might also get stuck in solving its module. Therefore, we use the same heuristic in the secondary solver to exit speculation.

While this is a simple heuristic, we found it to be useful in our benchmarks. The best strategy for speculation is problem-dependent. We leave development of a robust heuristic for future work.

# rounds	time (s) – sat		# rounds	time (s) – unsat	
	SMS	SPECSMS		SMS	SPECSMS
16	0.86	0.94	16	1.09	0.93
21	–	0.49	21	–	1.17
26	–	2.93	26	–	1.95
31	–	1.33	31	–	2.06
36	–	1.35	36	–	2.13
40	–	1.56	40	–	2.64

Table 1: Solving time with a timeout of 600s.

6 Implementation and Validation

We implemented SPECSMS (and SMS) inside the extensible SAT-solver of Z3 [5]⁴. For SMS, we simply disable speculation (Table 1).

We have validated SPECSMS on a set of handcrafted benchmarks, based on Ex. 1. Each benchmark is of the form $\langle \psi_{in}(\ell, in), \psi_{SHA-1}(in, out) \rangle$, where ℓ is a 2-bit vector, in is a 512-bit vector (shared), out is 160-bit vector. ψ_{in} encodes that there are four possible messages:

$$\begin{aligned} \psi_{in} \triangleq & (\ell = 0 \wedge in = msg_0) \vee (\ell = 1 \wedge in = msg_1) \vee \\ & (\ell = 2 \wedge in = msg_2) \vee (\ell = 3 \wedge in = msg_3) \end{aligned}$$

and $\psi_{SHA-1}(in, out)$ encodes the SHA-1 circuit together with some hash:

$$\psi_{SHA-1} \triangleq (SHA-1_{circ}(in) \wedge out = shaVal)$$

In the first set of experiments, we check sat queries by generating one msg_i in ψ_{in} that produces $shaVal$. In the second set, we check unsat queries, by ensuring that no msg_i produces $shaVal$. To evaluate performance, we make ψ_{SHA-1} harder to solve by increasing the number of rounds of SHA-1 circuit encoded in the $SHA-1_{circ}$ clauses. We used SAT-encoding [13]⁵ to generate the $SHA-1_{circ}$ with the different number of rounds (SAT-encoding supports 16 to 40 rounds).

We use the heuristic described in Sec. 5 to decide when to enter and exit speculation. Thus, SPECSMS switches modules when it hits too many conflicts in the module. In contrast, SMS only switches to the secondary solver after finding a full satisfying assignment in the main solver.

Results for each set of the queries are shown in Tab. 1. Column “# rounds” shows the number of SHA-1 rounds encoded in ψ_{SHA-1} . The problems quickly become too hard for SMS. At the same time, SPECSMS solves all the queries quickly. Furthermore, the run-time of SPECSMS appears to grow linearly with the number of rounds.

The experiments validate our claim that switching between modules is quite effective in solving the problem. As expected, SMS gets stuck in inverting the

⁴ we will provide the repository url after the double-blind review process

⁵ Available at <https://github.com/saeednj/SAT-encoding>.

SHA-1 function. It cannot make progress without using information from the secondary module. In contrast, SPECSMS switches to the secondary module once it finds that solving $\text{SHA-1}_{\text{circ}}(in)$ is hard. Note that, in this problem, the ideal strategy is to speculate eagerly and then branch on all the ℓ variables. However, SPECSMS spend some time solving $\text{SHA-1}_{\text{circ}}(in)$. It only switches to the secondary module when it hits many conflicts in $\text{SHA-1}_{\text{circ}}(in)$.

7 Conclusion and Future Work

Modular SAT-solving is crucial for efficient SAT-based unbounded Model Checking. Existing techniques, embedded in IC3/PDR [3] and extended in SMS [1], trade the efficiency of the solver for the simplicity of conflict resolution. In this paper, we propose a new modular SAT-solver, called SPECSMS, that extends SMS with truly bi-directional reasoning. We show that it is provably better than SMS (and, therefore, IC3/PDR). We implement SPECSMS in Z3 [5], extend it with DRUP-style [12] proofs, and proof-based interpolation. This work is an avenue to future efficient SAT- and SMT-based Model Checking algorithms.

In this paper, we rely on a simple heuristic to guide SPECSMS when to start speculation and exit speculation. This is sufficient to show the power of bi-directional reasoning over uni-directional reasoning on our benchmarks. However, other application domains might need more complicated heuristics to make this decision. In the future, we plan to explore guiding speculation using similar strategy used for guiding restarts in a modern CDCL SAT-solver[2].

A much earlier version of speculation, called *weak abstraction*, is implemented in the SPACER Constrained Horn Clause (CHC) solver [10]. Since SPACER extends IC3/PDR to SMT, the choice of speculation is based on theory reasoning. Speculation starts when the main solver is satisfied modulo some theories (e.g., Linear Real Arithmetic or Weak Theory of Arrays). Speculation often prevents SPACER from being stuck in any one SMT query. However, SPACER has no inter-modular propagation and no *refinement*. If *validation* fails, speculation is simply disabled and the query is tried again without it. We hope that extending SPECSMS to theories will make SPACER heuristics much more flexible and effective.

DPLL(T)-style [7] SMT-solvers can be seen as modular SAT-solvers where the main module is a SAT solver and the secondary solver is a theory solver (often EUF-solver that is connected to other theory solvers such as a LIA solver). This observation credited as an intuition for SMS [1]. In modern SMT-solvers, all decisions are made by the SAT-solver. For example, if a LIA solver wants to split on a bound of a variable x , it first adds a clause $(x \leq (b-1) \vee x \geq b)$, where b is the desired bound, to the SAT-solver and then lets the SAT-solver branch on the clause. SPECSMS extends this interaction by allowing the secondary solver (i.e., the theory solver) to branch without going back to the main solver. Control is returned to the main solver only if such decisions tangle local decisions of the two solvers. We hope that the core ideas of SPECSMS can be lifted to SMT and allow more flexibility in the interaction between the DPLL-core and theory solvers.

Acknowledgment The research leading to these results has received funding from the European Research Council under the European Union’s Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 2117/23. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), and MathWorks Inc. The first author was funded by Microsoft Research PhD Fellowship. The second author is not affiliated with the University of Waterloo at the time of publication.

References

1. S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu. Efficient modular SAT solving for IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 149–156. IEEE, 2013.
2. A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
3. A. R. Bradley. SAT-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
4. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.
5. L. M. de Moura and N. S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, 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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
6. N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
7. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): fast decision procedures. In R. Alur and D. A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.
8. A. V. Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008.
9. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposi-*

- tion (*DATE 2003*), 3-7 March 2003, Munich, Germany, pages 10886–10891. IEEE Computer Society, 2003.
10. A. Gurfinkel. Program verification with constrained horn clauses (invited paper). In S. Shoham and Y. Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 19–29. Springer, 2022.
 11. A. Gurfinkel and Y. Vizel. DRUPing for interpolates. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 99–106. IEEE, 2014.
 12. M. Heule, W. A. H. Jr., and N. Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.
 13. S. Nejati, J. H. Liang, V. Ganesh, C. Gebotys, and K. Czarnecki. Sha-1 preimage instances for sat. *SAT COMPETITION 2017*, page 45.

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.





Happy Ending: An Empty Hexagon in Every Set of 30 Points

Marijn J. H. Heule^{1,2}  and Manfred Scheucher³ 

¹ Carnegie Mellon University, Pittsburgh, USA
marijn@cmu.edu

² Amazon Scholar, Seattle, USA

³ Institute of Mathematics, Technische Universität Berlin, Berlin, Germany
scheucher@math.tu-berlin.de

Abstract. Satisfiability solving has been used to tackle a range of long-standing open math problems in recent years. We add another success by solving a geometry problem that originated a century ago. In the 1930s, Esther Klein’s exploration of unavoidable shapes in planar point sets in general position showed that every set of five points includes four points in convex position. For a long time, it was open if an empty hexagon, i.e., six points in convex position without a point inside, can be avoided. In 2006, Gerken and Nicolás independently proved that the answer is no. We establish the exact bound: Every 30-point set in the plane in general position contains an empty hexagon. Our key contributions include an effective, compact encoding and a search-space partitioning strategy enabling linear-time speedups even when using thousands of cores.

Keywords: Erdős–Szekeres problem · empty hexagon theorem · planar point set · cube-and-conquer · proof of unsatisfiability

1 Introduction

In 1932, Esther Klein showed that every set of five points in the plane *in general position* (i.e., no three points on a common line) has a subset of four points in convex position. Shortly after, Erdős and Szekeres [8] generalized this result by showing that, for every integer k , there exists a smallest integer $g(k)$ such that every set of $g(k)$ points in the plane in general position contains a k -gon (i.e., a subset of k points that form the vertices of a convex polygon). As the research led to the marriage of Szekeres and Klein, Erdős named it the *happy ending problem*. Erdős and Szekeres constructed witnesses of $g(k) > 2^{k-2}$ [9], which they conjectured to be maximal. The best upper bound is $g(k) \leq 2^{k+o(k)}$ [20, 30].

Determining the value $g(5) = 9$ requires a more involved case distinction compared to $g(4) = 5$ [23]. It took until 2006 to determine that $g(6) = 17$ via an exhaustive computer search by Szekeres and Peters [31] using 1500 CPU hours. Marić [25] and Scheucher [28] independently verified $g(6) = 17$ using satisfiability (SAT) solving in a few CPU hours. This was later reduced to 10 CPU minutes [29]. The approach presented in this paper computes it in 8.53 CPU seconds, showing the effectiveness of SAT compared to the original method.

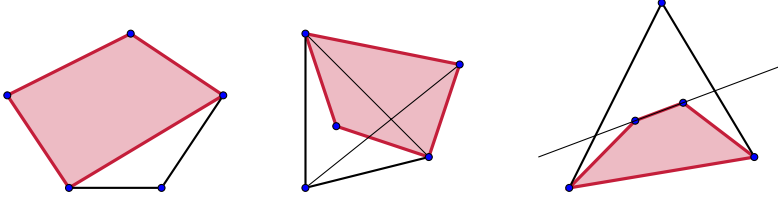


Fig. 1. An illustration for the proof of $h(4) = 5$: The three possibilities of how five points can be placed. Each possibility implies a 4-hole.

Erdős also asked whether every sufficiently large point set contains a *k-hole*: a *k*-gon without a point inside. We denote by $h(k)$ the smallest integer—if it exists—such that every set of $h(k)$ points in general position in the plane contains a *k-hole*. Both $h(3) = 3$ and $h(4) = 5$ are easy to compute (see Fig. 1 for an illustration) and coincide with the original setting. Yet the answer can differ a lot, as Horton [21] constructed arbitrarily large point sets without 7-holes.

While Harborth [14] showed in 1978 that $h(5) = 10$, the existence of 6-holes remained open until the late 2000s, when Gerken [12]⁴ and Nicolás [26] independently proved that $h(6)$ is finite. Gerken proved that every 9-gon yields a 6-hole, thereby showing that $h(6) \leq g(9) \leq 1717$ [33]. The best-known lower bound $h(6) \geq 30$ is witnessed by a set of 29 points without 6-holes which was found by Overmars [27] using a local search approach.

We close the gap between the upper and lower bound and ultimately answer Erdős’ question by proving that every set of 30 points yields a 6-hole.

Theorem 1. $h(6) = 30$.

Our result is actually stronger and shows that the bounds for 6-holes in point sets coincide with the bounds for 6-holes in *counterclockwise systems* [24]. This represents another success of solving long-standing open problems in mathematics using SAT, similar to results on Schur number five [16] and Keller’s conjecture [4].

We also investigate the combination of 6-holes and 7-gons and show

Theorem 2. *Every set of 24 points in the plane in general position contains a 6-hole or a 7-gon.*

We achieve these results through the following contributions:

- We develop a compact and effective SAT encoding for *k*-gon and *k*-hole problems that uses $O(n^4)$ clauses, while existing encodings use $O(n^k)$ clauses.
- We construct a partitioning of *k*-gon and *k*-hole problems that allows us to solve them with linear-time speedups even when using thousands of cores.
- We present a novel method of validating SAT-solving results that checks the proof while solving the problem using substantially less overhead.
- We verify most of the presented results using clausal proof checking.

⁴ Gerken’s groundbreaking work was awarded the Richard-Rado prize by the German Mathematical Society in 2008.

2 Preliminaries

The SAT problem. The satisfiability problem (SAT) asks whether a Boolean formula can be satisfied by some assignment of truth values to its variables. The Handbook of Satisfiability [2] provides an overview. We consider formulas in *conjunctive normal form* (CNF), which is the default input of SAT solvers. As such, a formula Γ is a conjunction (logical “AND”) of *clauses*. A clause is a disjunction (logical “OR”) of literals, where a literal is a Boolean variable or its negation. We sometimes write (sets of) clauses using other logical connectives.

If a formula Γ is found to be satisfiable, modern SAT solvers commonly output a truth assignment of the variables. Additionally, if a formula turns out to be unsatisfiable, sequential SAT solvers produce an independently-checkable proof that there exists no assignment that satisfies the formula.

Verification. The most commonly-used proofs for SAT problems are expressed in the DRAT clausal proof system [15]. A DRAT proof of unsatisfiability is a list of clause addition and clause deletion steps. Formally, a clausal proof is a list of pairs $\langle s_1, C_1 \rangle, \dots, \langle s_m, C_m \rangle$, where for each $i \in \{1, \dots, m\}$, $s_i \in \{\mathbf{a}, \mathbf{d}\}$ and C_i is a clause. If $s_i = \mathbf{a}$, the pair is called an *addition*, and if $s_i = \mathbf{d}$, it is called a *deletion*. For a given input formula Γ_0 , a clausal proof gives rise to a set of *accumulated formulas* Γ_i ($i \in \{1, \dots, m\}$) as follows:

$$\Gamma_i = \begin{cases} \Gamma_{i-1} \cup \{C_i\} & \text{if } s_i = \mathbf{a} \\ \Gamma_{i-1} \setminus \{C_i\} & \text{if } s_i = \mathbf{d} \end{cases}$$

Each clause addition must preserve satisfiability, which is usually guaranteed by requiring the added clauses to fulfill some efficiently decidable syntactic criterion. Deletions help to speed up proof checking by keeping the accumulated formula small. A valid proof of unsatisfiability must add the empty clause.

Cube And Conquer. The cube-and-conquer approach [18] aims to *split* a SAT instance Γ into multiple instances $\Gamma_1, \dots, \Gamma_m$ in such a way that Γ is satisfiable if and only if at least one of the instances Γ_i is satisfiable, thus allowing work on the different instances Γ_i in parallel. A *cube* is a conjunction of literals. Let $\psi = (c_1 \vee \dots \vee c_m)$ be a disjunction of cubes. When ψ is a tautology, we have

$$\Gamma \iff \Gamma \wedge \psi \iff \bigvee_{i=1}^m (\Gamma \wedge c_i) \iff \bigvee_{i=1}^m \Gamma_i,$$

where the different $\Gamma_i := (\Gamma \wedge c_i)$ are the instances resulting from the split.

Intuitively, each cube c_i represents a *case*, i.e., an assumption about a satisfying assignment to Γ , and soundness comes from ψ being a tautology, which means that the split into cases is exhaustive. If the split is well designed, then each Γ_i is a particular case that is substantially easier to solve than Γ , and thus solving them all in parallel can give significant speed-ups, especially considering the sequential nature of CDCL at the core of most solvers.

However, the quality of the split (ψ) has an enormous impact on the effectiveness of the approach. A key challenge is figuring out a high-quality split.

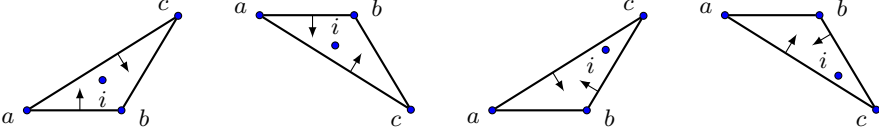


Fig. 2. The four ways a point p_i can be inside triangle $\{p_a, p_b, p_c\}$ based on whether $i < b$ (left two images) and whether p_c is above the line p_ap_b (first and third image).

3 Trusted Encoding

To obtain an upper-bound result using a SAT-based approach, we need to show that every set of n points contains a k -hole. We will do this by constructing a formula based on n points that asks whether a k -hole can be avoided. If this formula is unsatisfiable, then we obtain the bound $h(k) \leq n$. Instead of reasoning directly whether an empty k -gon can be avoided, we ask whether every k points contain at least one triangle with a point inside. The latter implies the former.

We only need to know for each triple of points whether it is empty. Throughout the paper, we assume that points are sorted with strictly increasing x -coordinates. This gives us only four options for a point p_i to be inside the triangle formed by points p_a, p_b, p_c , see Fig. 2. For example, the left image shows that p_i is inside if $a < i < b$, p_c and p_i are above the line $\overline{p_ap_b}$, and p_i is below the line $\overline{p_ap_c}$. So we need some machinery to express that points are above or below certain lines. That is what the encoding will provide. For readability, we sometimes identify points by their indices, that is, we refer to p_a by its index a .

We first present what we call the *trusted encoding* to determine whether a 6-hole can be avoided. The encoding needs to be trusted in the sense that we do not provide a mechanically verified proof of its correctness. Building upon existing work [28], our primary focus is on 6-holes, which constitute our main result. The encoding of 6-gons and 7-gons is similar and more simple. During an initial study, the estimated runtime for showing $h(6) \leq 30$ using this encoding and off-the-shelf partitioning was roughly 1000 CPU years. The optimizations in Sections 4 and 5 reduce the computational costs to about 2 CPU years.

3.1 Orientation Variables

We formulate the problem in such a way that all reasoning is based solely on the relative positions of points. Thus, we do not encode coordinates but only orientations of point triples. For a point set $S = \{p_1, \dots, p_n\}$ with $p_i = (x_i, y_i)$, the triple (p_a, p_b, p_c) with $a < b < c$ is *positively oriented* (resp. *negatively oriented*) if p_c lies above (resp. below) the line $\overline{p_ap_b}$ through p_a and p_b . The notion of positive orientation corresponds to Knuth's *counterclockwise relation* [24]. Fig. 3 illustrates a positively-oriented triple (p_a, p_b, p_c) and a negatively-oriented triple (p_a, p_b, p_d) .

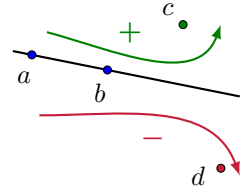


Fig. 3. An illustration of triple orientations.

To search for point sets without k -gons and k -holes, we introduce a Boolean *orientation variable* $\mathbf{o}_{a,b,c}$ for each triple (p_a, p_b, p_c) with $a < b < c$. Intuitively, $\mathbf{o}_{a,b,c}$ is supposed to be true if the triple is positively oriented. Since we assume general position, no three points lie on a common line, so $\mathbf{o}_{a,b,c}$ being false means that the triple is negatively oriented.

3.2 Containment Variables, 3-Hole Variables, and Constraints

Using orientation variables, we can now express what it means for a triangle to be empty. We define *containment variables* $\mathbf{c}_{i;a,b,c}$ to encode whether point p_i lies inside the triangle spanned by $\{p_a, p_b, p_c\}$. Since the points have increasing x -coordinates, containment is only possible if $a < i < c$. We use two kinds of definitions, depending on whether i is smaller or larger than b (see Fig. 2). The first definition is for the case $a < i < b$. Note that if $\mathbf{o}_{a,b,c}$ is true, we only need to know whether i is above the line $\overline{p_a p_b}$ and below the line $\overline{p_a p_c}$. Earlier work [28] used an extended definition that included the redundant variable $\mathbf{o}_{i,b,c}$. Avoiding this variable makes the definition more compact (six instead of eight clauses) and the resulting formula is easier to solve.

$$\mathbf{c}_{i;a,b,c} \leftrightarrow \left((\mathbf{o}_{a,b,c} \rightarrow (\overline{\mathbf{o}_{a,i,b}} \wedge \mathbf{o}_{a,i,c})) \wedge (\overline{\mathbf{o}_{a,b,c}} \rightarrow (\mathbf{o}_{a,i,b} \wedge \overline{\mathbf{o}_{a,i,c}})) \right) \quad (1)$$

The second definition is for $b < i < c$, which avoids using the variable $\mathbf{o}_{a,b,i}$:

$$\mathbf{c}_{i;a,b,c} \leftrightarrow \left((\mathbf{o}_{a,b,c} \rightarrow (\mathbf{o}_{a,i,c} \wedge \overline{\mathbf{o}_{b,i,c}})) \wedge (\overline{\mathbf{o}_{a,b,c}} \rightarrow (\overline{\mathbf{o}_{a,i,c}} \wedge \mathbf{o}_{b,i,c})) \right) \quad (2)$$

Each definition translates into six clauses (without using Tseitin variables).

Additionally, we introduce definitions $\mathbf{h}_{a,b,c}$ of 3-hole variables that express whether the triangle spanned by $\{p_a, p_b, p_c\}$ is a 3-hole. The triangle $\{p_a, p_b, p_c\}$ forms a 3-hole if and only if no point p_i lies in its interior. A point p_i can only be an inner point if it lies in the vertical strip between p_a and p_c and if it is distinct from p_b . Since the points are sorted, the index i of an interior point p_i must therefore fulfill $a < i < c$ and $i \neq b$. Logically, the definition is as follows:

$$\mathbf{h}_{a,b,c} \leftrightarrow \bigwedge_{\substack{a < i < c \\ i \neq b}} \overline{\mathbf{c}_{i;a,b,c}}. \quad (3)$$

Finally, we encode the “forbid k -hole” constraint as follows: For each subset $X \subseteq S$ of size k , at least one of the triangles formed by three points in X must not be a 3-hole. So for $k = 6$, each clause consists of $\binom{k}{3} = 20$ literals.

$$\bigwedge_{\substack{X \subseteq S \\ |X|=k}} \left(\bigvee_{\substack{a,b,c \in X \\ a < b < c}} \overline{\mathbf{h}_{a,b,c}} \right) \quad (4)$$

In Section 4, we will optimize the encoding. Most optimizations aim to improve the encoding of the constraint (4).

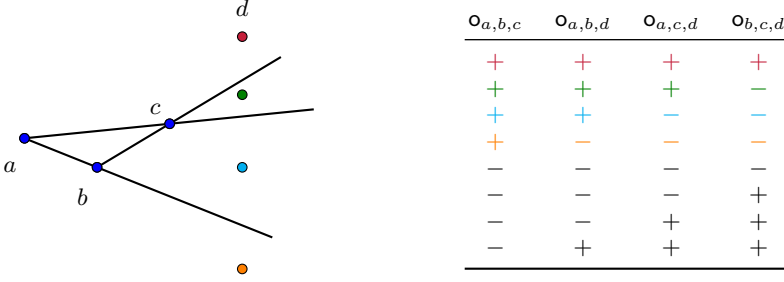


Fig. 4. All possibilities to place four points, when points are sorted from left to right.

3.3 Forbidding Non-Realizable Patterns

Only a small fraction of all assignments to the $\binom{n}{3}$ orientation variables, $2^{\Theta(n \log n)}$, actually describe point sets [3]. However, we can reduce the search space from $2^{\Theta(n^3)}$ to $2^{\Theta(n^2)}$ by forbidding non-realizable patterns [24]. Consider four points p_a, p_b, p_c, p_d in a sorted point set with $a < b < c < d$. The leftmost three points determine three lines $\overline{p_a p_b}$, $\overline{p_a p_c}$, $\overline{p_b p_c}$, which partition the open half-plane $\{(x, y) \in \mathbb{R}^2 : x > x_c\}$ into four regions (see Fig. 4). After placing p_a, p_b, p_c , observe that all realizable positions of point p_d obey the following implications: $o_{a,b,c} \wedge o_{a,c,d} \Rightarrow o_{a,b,d}$ and $o_{a,b,c} \wedge o_{b,c,d} \Rightarrow o_{a,c,d}$. Similarly for the negations, $\overline{o_{a,b,c}} \wedge \overline{o_{a,c,d}} \Rightarrow \overline{o_{a,b,d}}$ and $\overline{o_{a,b,c}} \wedge \overline{o_{b,c,d}} \Rightarrow \overline{o_{a,c,d}}$. These implications are equivalent to the following clauses (grouping positive and negative):

$$(\overline{o_{a,b,c}} \vee \overline{o_{a,c,d}} \vee o_{a,b,d}) \wedge (o_{a,b,c} \vee o_{a,c,d} \vee \overline{o_{a,b,d}}) \quad (5)$$

$$(\overline{o_{a,b,c}} \vee \overline{o_{b,c,d}} \vee o_{a,c,d}) \wedge (o_{a,b,c} \vee o_{b,c,d} \vee \overline{o_{a,c,d}}) \quad (6)$$

Forbidding these non-realizable assignments was also used for $g(6) \leq 17$ [31]. Some call the restriction *signotope axioms* [10]. The counterclockwise system axioms [24] achieve the same effect, but require $\Theta(n^5)$ clauses instead of $\Theta(n^4)$.

3.4 Initial Symmetry Breaking

To further reduce the search space, we ensure that p_1 lies on the boundary of the convex hull (i.e., it is an extremal point) and that p_2, \dots, p_n appear around p_1 in counterclockwise order, thus providing us the unit clauses $(o_{1,a,b})$ for $1 < a < b$. Without loss of generality, we can label points to satisfy the above, because the labeling doesn't affect gons and holes. However, we also want points to be sorted from left to right. One can satisfy both orderings at the same time using the lemma below. We attach a proof in the extended version [19].

Lemma 1 ([28, Lemma 1]). *Let $S = \{p_1, \dots, p_n\}$ be a point set in the plane in general position such that p_1 is extremal and p_2, \dots, p_n appear (clockwise or counterclockwise) around p_1 . Then there exists a point set $\tilde{S} = \{\tilde{p}_1, \dots, \tilde{p}_n\}$ with the same triple orientations (in particular, \tilde{p}_1 is extremal and $\tilde{p}_2, \dots, \tilde{p}_n$ appear around \tilde{p}_1) such that the points $\tilde{p}_1, \dots, \tilde{p}_n$ have increasing x -coordinates.*

4 Optimizing the Encoding

An ideal SAT encoding has the following three properties:

- 1) it is compact to reduce the cost of unit propagation (and cache misses);
- 2) it detects conflicts as early as possible (i.e., is domain consistent [11]); and
- 3) it contains variables that can generalize conflicts effectively.

The trusted encoding lacks these properties because it has $O(n^6)$ clauses, cannot quickly detect holes, and has no variables that can generalize conflicts. In this section, we show how to modify the trusted encoding to obtain all three properties. All the modifications are expressible in a proof to ensure correctness.

4.1 Toward Domain Consistency

The effectiveness of an encoding depends on how quickly the solver can determine a conflict. Given an assignment, we want to derive as much as possible via unit propagation. This is known as *domain consistency* [11]. The trusted encoding does not have this property. We modify the encoding below to boost propagation.

We borrow from the method by Szekeres and Peters that a k -gon can be detected by looking at assignments to $k - 2$ orientation variables [31]. For example, if $\mathbf{o}_{a,b,c}$, $\mathbf{o}_{b,c,d}$, $\mathbf{o}_{c,d,e}$, and $\mathbf{o}_{d,e,f}$ with $a < b < c < d < e < f$ are assigned to the same truth value, then this implies that the points form a 6-gon. An illustration of this assignment is shown in Fig. 5 (left). We combine this with our observation below that only a specific triangle has to be empty to infer a 6-hole somewhere.

Consider a scenario involving six points, a, b, c, d, e , and f , that are arranged from left to right. In this scenario, the orientation variables $\mathbf{o}_{a,b,c}$, $\mathbf{o}_{b,c,d}$, $\mathbf{o}_{c,d,e}$, and $\mathbf{o}_{d,e,f}$ are all set to false, while the 3-hole variable $\mathbf{h}_{a,c,e}$ is set to true. As mentioned above, this implies that the points form a 6-gon. Together with 3-hole variable $\mathbf{h}_{a,c,e}$ being set to true, we can deduce the existence of a 6-hole: The 6-gon is either a 6-hole or it contains a 6-hole. The reasoning will be explained in the next paragraph. Note that in the trusted encoding of this scenario, only one out of the twenty literals in the corresponding ‘forbid 6-hole’ clause is false. This suggests that the solver is still quite far from detecting a conflict.

A crucial insight underpinning our efficient encoding is the understanding that the truth of the variable $\mathbf{h}_{a,c,e}$ alone is sufficient to infer the existence of a 6-hole. Consider the following rationale: If the triangle $\{a, b, c\}$ contains any points, then there must be at least one point inside the triangle that is closer to the line \overline{ac} than point b is. Let’s denote the nearest point as i . The proximity of i to the line \overline{ac} guarantees that the triangle $\{a, i, c\}$ is empty. We can substitute b with i to create a smaller but similarly shaped hexagon. This logic extends to other triangles as well; specifically, the truth values of $\mathbf{h}_{c,d,e}$ and $\mathbf{h}_{a,e,f}$ are not necessary to infer the presence of a 6-hole.

Our insight emerged when we noticed that the SAT solver eliminated 3-hole literals from previous encodings. This elimination occurred primarily when only a few points existed between the leftmost and rightmost points of a triangle. On

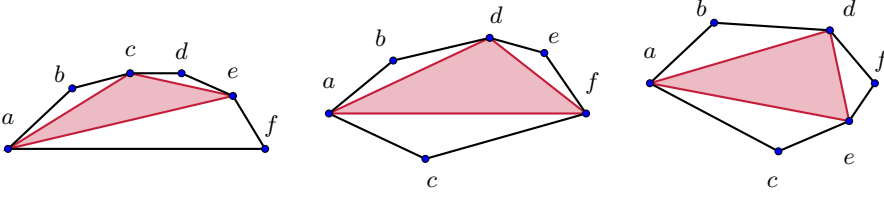


Fig. 5. Three types of 6-gons: left, all points are on one side of line \overline{af} (2 cases); middle, three points are on one side and one point is on the other side of line \overline{af} (8 cases); and right, two points are on either side of line \overline{af} (6 cases). If the marked triangle is empty, we can conclude that there exists a 6-hole.

the other hand, the solver struggles significantly to identify the redundancy of these 3-hole literals when the leftmost and rightmost points of a triangle were far apart. Therefore, to enhance the encoding’s effectiveness, we chose to omit these 3-hole literals (instead of letting the solver figure it out).

Blocking the existence of a 6-hole within the 6-gon described above can be achieved with the following clause (which simply negates the assignment):

$$o_{a,b,c} \vee o_{b,c,d} \vee o_{c,d,e} \vee o_{d,e,f} \vee \overline{h_{a,c,e}} \quad (7)$$

For each set of six points, 16 different configurations can result in a 6-hole. These configurations depend on which points are positioned left or right the line connecting the leftmost and rightmost points among the six. The three types of such configurations are illustrated in Fig. 5, while the remaining configurations are symmetrical to these. It is important to note that this adds $16 \times \binom{n}{6}$ clauses to the formula, significantly increasing its size.

We can reduce the number of clauses by about a 30% by strategically selecting which triangle within a 6-gon is checked to be empty (i.e., which 3-hole literal will be used). The two options are the triangle that includes the leftmost point (as depicted in Fig. 5) and the triangle with the second-leftmost point. If the leftmost point is p_1 , we opt for the second-leftmost point; otherwise, we choose the leftmost point. After propagating the unit clauses $o_{1,a,b}$, the clauses that describe configurations with three points below the line \overline{af} become subsumed by the clause for the configuration with four points below the line $\overline{1f}$.

4.2 An $O(n^4)$ Encoding

This section is rather technical. It introduces auxiliary variables to reduce our encoding to $O(n^4)$ clauses. The process is known as structured bounded variable addition (SBVA) [13], which in each step adds a new auxiliary variable to encode a subset of the formula more compactly. SBVA heuristically selects the auxiliary variables. Instead, we select them manually because it is more effective, the new variables have meaning, and SBVA is extremely slow on this problem. Eliminating the auxiliary variables results in the encoding of Section 4.1.

The first type of these variables, $u_{a,c,d}^4$, represents the presence of a 4-gon $\{a, b, c, d\}$ such that points a, b, c, d appear in this order from left to right and b and c are above the line \overline{ad} . Furthermore, the variables $u_{a,d,e}^5$ indicate the existence of a 5-gon $\{a, b, c, d, e\}$ with the property that the points a, b, c, d, e appear in this order from left to right, the points b, c , and d are above the line \overline{ae} , and the triangle $\{a, c, e\}$ is empty. This configuration implies the existence of a 5-hole within $\{a, b, c, d, e\}$ using similar reasoning as described in Section 4.1. The clauses enforcing these properties are outlined below.

$$u_{a,c,d}^4 \vee o_{a,b,c} \vee o_{b,c,d} \quad \text{with } a < b < c < d \quad (8)$$

$$u_{a,d,e}^5 \vee \overline{u_{a,c,d}^4} \vee o_{c,d,e} \vee \overline{h_{a,c,e}} \quad \text{with } a < c < d < e \quad (9)$$

In the following we distinguish five types of 6-holes by the number of its points that lie above/below the line connecting its leftmost and rightmost points. Fig. 5 shows the three configurations with four, three, and two points above the line, respectively. The two cases with three and four points below the line are symmetric but will be handled in a different and more efficient manner below.

To block all 6-holes with configurations having three or four points above the line connecting the leftmost and rightmost points, we utilize the variables $u_{a,d,e}^5$. Specifically, a configuration with three points above occurs if there is a point b situated between a and e , lying below the line \overline{ae} . Also, the configuration with four points above arises when a point f , located to the right of e , falls below the line \overline{de} . The associated clauses for these configurations are detailed below. The omission of 3-hole literals is justified by our knowledge that a 3-hole exists among a, c , and e for some point c positioned above the line \overline{ae} .

$$\overline{u_{a,d,e}^5} \vee \overline{o_{a,b,e}} \quad \text{with } a < d < e, a < b < e \quad (10)$$

$$\overline{u_{a,d,e}^5} \vee o_{d,e,f} \quad \text{with } a < d < e < f \quad (11)$$

To block the third type of a 6-hole, we need to introduce variables $v_{a,c,d}^4$ which, similar as $u_{a,c,d}^4$, indicate the presence of a 4-gon $\{a, b, c, d\}$ with the property that the points a, b, c, d appear in this order from left to right and b and c are *below* the line \overline{ad} . The clauses that encode these variables are:

$$v_{a,c,d}^4 \vee \overline{o_{a,b,c}} \vee \overline{o_{b,c,d}} \quad \text{with } a < b < c < d \quad (12)$$

Using the variables $u_{a,c,d}^4$ and $v_{a,c',d}^4$ we are now ready to block the configuration of the third type of a 6-hole where two points lie above and two points lie below the line connecting the leftmost and rightmost points; see Fig. 5 (right). Recall that $u_{a,c,d}^4$ denotes a 4-gon situated above the line \overline{ad} , with c being the second-rightmost point. Also, $v_{a,c',d}^4$ denotes a 4-gon below the line \overline{ad} , with c' as the second-rightmost point. A 6-hole exists if both $u_{a,c,d}^4$ and $v_{a,c',d}^4$ are true for some points a and d when there are no points within the triangle formed by a, c , and c' . Or, in clauses:

$$\overline{u_{a,c,d}^4} \vee \overline{v_{a,c',d}^4} \vee \overline{h_{a,c,c'}} \quad \text{with } a < c < c' < d \quad (13)$$

$$u_{a,c,d}^4 \vee v_{a,c',d}^4 \vee \overline{h_{a,c',c}} \quad \text{with } a < c' < c < d \quad (14)$$

The remaining configurations to consider involve those with three or four points below the line joining the leftmost and rightmost points. As we discussed at the end of Section 4.1, these configurations can be encoded more compactly. We only need to block the existence of 5-holes $\{a, b, c, d, e\}$ with the property that the points $1, a, b, c, d, e$ appear in this order from left to right and the points b, c , and d are below the line \overline{ae} . The reasoning is as follows: if such a 5-hole exists, it can be expanded into a 6-hole by the closest point to line \overline{ab} within the triangle $\{1, a, b\}$ (which is point 1 if the triangle is empty). Additionally, by blocking these specific 5-holes, we simultaneously block all 6-holes with three or four points below the line between the leftmost and rightmost points. Following the earlier cases, we only require a single 3-hole literal which ensures that the triangle $\{a, c, e\}$ is empty. The clauses to block these 5-holes are as follows:

$$\overline{v_{a,c,d}^4} \vee \overline{o_{c,d,e}} \vee \overline{h_{a,c,e}} \quad \text{with } 1 < a < c < d < e \quad (15)$$

This encoding uses $O(n^4)$ clauses, while it has the same propagation power as having all the $16 \times \binom{n}{6}$ clauses in the domain-consistent encoding of Section 4.1. In general, the trusted encoding for k -holes uses $O(n^k)$ clauses, while the optimized encoding when generalized to k -holes has only $O(kn^4)$ clauses, or $O(n^4)$ for every fixed k . An encoding of size $O(n^4)$ for k -gons is analogous: simply remove the 3-hole literals from the clauses.

4.3 Minor Optimizations

We can make the encoding even more compact by removing a large fraction of the clauses from the trusted encoding. Note that constraints to forbid 6-holes contain only negative 3-hole literals. That means that only half of the constraints to define the 3-hole variables are actually required. This in turn shows that only half of the inside variable definitions are required. So, instead of (1), (2), and (3), it suffices to use the following:

$$c_{i;a,b,c} \rightarrow \left((o_{a,b,c} \rightarrow (\overline{o_{a,i,b}} \wedge o_{a,i,c})) \wedge (\overline{o_{a,b,c}} \rightarrow (o_{a,i,b} \wedge \overline{o_{a,i,c}})) \right) \quad (16)$$

$$c_{i;a,b,c} \rightarrow \left((o_{a,b,c} \rightarrow (o_{a,i,c} \wedge \overline{o_{b,i,c}})) \wedge (\overline{o_{a,b,c}} \rightarrow (\overline{o_{a,i,c}} \wedge o_{b,i,c})) \right) \quad (17)$$

$$h_{a,b,c} \leftarrow \bigwedge_{\substack{a < i < c \\ i \neq b}} \overline{c_{i;a,b,c}}. \quad (18)$$

It is worth noting that the SAT preprocessing technique blocked-clause elimination (BCE) can automatically remove the omitted clauses [22]. However, for means of efficiency, BCE is turned off by default in top-tier solvers, including the solver CaDiCaL, which we used for the proof. During initial experiments, we observed that omitting these clauses slightly improves the performance.

Finally, the variables $u_{a,c,d}^4$ and $v_{a,c,d}^4$ can be used to more compactly encode the clauses (6). We can replace them with the following clauses:

$$(\overline{u_{a,c,d}^4} \vee \overline{o_{a,c,d}}) \wedge (\overline{v_{a,c,d}^4} \vee o_{a,c,d}) \quad \text{with } a < c < d \quad (19)$$

4.4 Breaking the Reflection Symmetry

Holes are invariant to reflectional symmetry: If we mirror a point set S , then the counterclockwise order around the extremal point p_1 (which is p_2, \dots, p_n) is reversed (to p_n, \dots, p_2). By relabeling points to preserve the counterclockwise order, we preserve $\mathbf{o}_{1,a,b} = \text{true}$ for $a < b$, while the original orientation variables $\mathbf{o}_{a,b,c}$ with $2 \leq a < b < c \leq n$ are mapped to $\mathbf{o}_{n-c+2,n-b+2,n-a+2}$. A similar mapping applies to the containment and 3-hole variables. The trusted encoding maps almost onto itself, except for the missing reflection clauses of (5) and (6). As a fix for verification, we add each reflected clause using one resolution step.

Since only a tiny fraction of triple orientations map to themselves (so-called *involutions*), breaking the reflectional symmetry reduces the search space by a factor of almost 2. We partially break this symmetry by constraining the variables $\mathbf{o}_{a,a+1,a+2}$ with $2 \leq a \leq n-2$. We used the symmetry-breaking predicate below, because it is compatible with our cube generation, described in Section 5.

$$\mathbf{o}_{\lceil \frac{n}{2} \rceil - 1, \lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil + 1, \dots, \mathbf{o}_{2,3,4} \preceq \mathbf{o}_{\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \lfloor \frac{n}{2} \rfloor + 3, \dots, \mathbf{o}_{n-2, n-1, n} \quad (20)$$

One symmetry that remains is the choice of the first point. Any point on the convex hull could be picked for this purpose, and breaking it can potentially reduce the search space by at least a factor of 3. However, breaking this symmetry effectively is complicated and we therefore left it on the table.

5 Problem Partitioning

The formula to determine that $h(6) \leq 30$ requires CPU years to solve. To compute this in reasonable time, the problem needs to be partitioned into many small subproblems that can be solved in parallel. Although there exist tools to do the partitioning automatically [18], we observed that this partitioning was ineffective. As a consequence, we focused on manual partitioning.

During our initial experiments, we determined which orientation variables were suitable for splitting. We used the formula for $g(6) \leq 17$ for this purpose because its runtime is large enough to make meaningful observations and small enough to explore many options. It turned out that the variables $\mathbf{o}_{a,a+1,a+2}$ were the most effective choice for splitting the problem. Assigning one of these $\mathbf{o}_{a,a+1,a+2}$ variables to true/false roughly halves the search space and reduces the runtime by a factor of roughly 2.

A problem with n points has $n-3$ free variables of the form $\mathbf{o}_{a,a+1,a+2}$, as the variable $\mathbf{o}_{1,2,3}$ is already fixed by the symmetry breaking. One cannot generate 2^{n-3} equally easy subproblems, because $(\overline{\mathbf{o}_{a,a+1,a+2}} \vee \overline{\mathbf{o}_{a+1,a+2,a+3}} \vee \overline{\mathbf{o}_{a+2,a+3,a+4}})$ and $(\mathbf{o}_{a,a+1,a+2} \vee \mathbf{o}_{a+1,a+2,a+3} \vee \mathbf{o}_{a+2,a+3,a+4} \vee \mathbf{o}_{a+3,a+4,a+5})$ follow directly from the optimized formula after unit propagation. Thus, assigning three consecutive $\mathbf{o}_{a,a+1,a+2}$ variables to true results directly in a falsified clause, as it would create a 6-hole among the points p_1, p_a, \dots, p_{a+4} . The same holds for four consecutive $\mathbf{o}_{a,a+1,a+2}$ variables assigned to false, which would create a 6-hole among the

points p_a, \dots, p_{a+5} . The asymmetry is due to fixing the variables $\mathbf{o}_{1,a,b}$ to true. If we assigned them to false, then the opposite would happen.

We observed that limiting the partition to variables involving the middle points reduces the total runtime. We will demonstrate such experiments in Section 6.2. So, to obtain suitable cubes, we considered all assignments of the sequence $\mathbf{o}_{a,a+1,a+2}, \mathbf{o}_{a+1,a+2,a+3}, \dots, \mathbf{o}_{a+\ell-1,a+\ell,a+\ell+1}$ for a suitable constant ℓ and $a = \frac{n+\ell}{2} - 1$ such that the above properties are fulfilled, that is, no three consecutive entries are true and no four consecutive entries are false. In the following we refer to ℓ as the *length* of the cube-space. In our experiments, we observed that picking $\ell < n - 3$ reduces the overall computational costs. Specifically, for the $h(6) \leq 30$ experiments, we use length $\ell = 21$.

Our initial experiments showed that the runtime of cubes grows exponentially with the number of occurrences of the alternating pattern $\mathbf{o}_{b,b+1,b+2} = +$, $\mathbf{o}_{b+1,b+2,b+3} = -$, $\mathbf{o}_{b+2,b+3,b+4} = +$. As a consequence, the hardest cube for $h(6) \leq 30$ would still require days of computing time, thereby limiting parallelism. To deal with this issue, we further partition cubes that contain this pattern. For each occurrence of the alternating pattern in a cube, we split the cube into two cubes: one that extends it with $\mathbf{o}_{b,b+2,b+4}$ and one that extends it with $\overline{\mathbf{o}_{b,b+2,b+4}}$. Note that we do this for each occurrence. So a cube containing m of these patterns is split into 2^m cubes. This reduced the computational costs of the hardest cubes to less than an hour.

6 Evaluation

For the experiments, we use the solver CaDiCaL (version 1.9.3) [1], which is currently the only top-tier solver that can produce LRAT proofs directly. The efficient, verified checker cakeLPR [32] validated the proofs. We run CaDiCaL with command-line options: `--sat --reducetarget=10 --forcephase --phase=0`. The first option reduces the number of restarts. This is typically more useful for satisfiable formulas (as the name suggests), but in this case it is also helpful for unsatisfiable formulas. The second option turns off the aggressive clause deletion strategy. The last two options turn on negative branching, a MiniSAT heuristic [7]. Experiments were run on a specialized, internal Amazon Web Services solver framework that provides cloud-level scaling. The framework used `m6i.xlarge` instances, which have two physical cores and 16 GB of memory.

6.1 Impact of the Encoding

To illustrate the impact of the encoding on the performance, we show some statistics on various encodings of the $h(6) \leq 30$ formula. We restricted this experiment to solving a single randomly-picked subproblem. For other subproblems, the results were similar. We experimented with the following five encodings:

- T : the trusted encoding presented in Section 3
- O_1 : T with (4) replaced by the domain-consistent encoding (7) of Section 4.1

- O_2 : O_1 with (7) replaced by the $O(n^4)$ encoding (8) - (15) of Section 4.2
- O_3 : O_2 with the minor optimizations that replace (1), (2), (3), and (6) by (17), (18), (18), and (19), respectively, see Section 4.3
- O_4 : O_3 extended with the symmetry-breaking predicate from Section 4.4

Table 1 summarizes the results. The domain-consistent encoding can be solved more efficiently than the trusted encoding while having over five times as many clauses. The reason for the faster performance becomes clear when looking at the number of conflicts and propagations. The domain-consistent encoding requires just over a fifth as many conflicts and propagations to determine unsatisfiability. The auxiliary variables that enable the $O(n^4)$ encoding reduce the size by almost an order of magnitude. The resulting formula can be solved three times as fast, while using a similar number of conflicts and propagations. The minor optimizations reduce the size by roughly a third and further improve the runtime. Finally, the addition of the symmetry-breaking predicate doesn't impact the performance. Its main purpose is to halve the number of cubes.

We also solved the optimized encoding (O_3) of the formula $g(6) \leq 17$, which takes 41.99 seconds using 623 540 conflicts. Adding the symmetry-breaking predicate (O_4) reduces the runtime to 17.39 seconds using 316 785 conflicts. So the symmetry-breaking predicate reduces the number of conflicts by roughly a factor of 2 (as expected) while the runtime is reduced even more. The latter is due to the slowdown caused by maintaining more conflict clauses while solving the formula without the symmetry-breaking predicate.

6.2 Impact of the Partitioning

All known point sets witnessing the lower bound $h(6) \geq 30$ contain a 7-gon. To obtain a possibly easier problem to test and compare heuristics, we studied how many points are required to guarantee the existence of a 6-hole or a 7-gon. It turned out that the answer is at most 24 (Theorem 2). Computing this is still hard but substantially easier compared to our main result. During our experiments, we observed that increasing the number of cubes can increase the total runtime. We therefore explored which parameters produce the lowest total runtime. The experimental results are shown in Table 2 for various values for the parameter ℓ . Incrementing ℓ by 2 increases the number of cubes roughly by a factor of 3. The optimal total runtime is achieved for $\ell = 15$, which is a 62%

Table 1. Comparison of the different encodings.

formula	#variables	#clauses	#conflicts	#propagations	time (s)
T	62 930	1 171 942	1 082 569	1 338 662 627	243.07
O_1	62 930	5 823 078	228 838	282 774 472	136.20
O_2	75 110	667 005	211 272	343 388 591	45.49
O_3	75 110	436 047	234 755	340 387 692	39.46
O_4	75 110	444 238	234 587	342 904 580	39.41

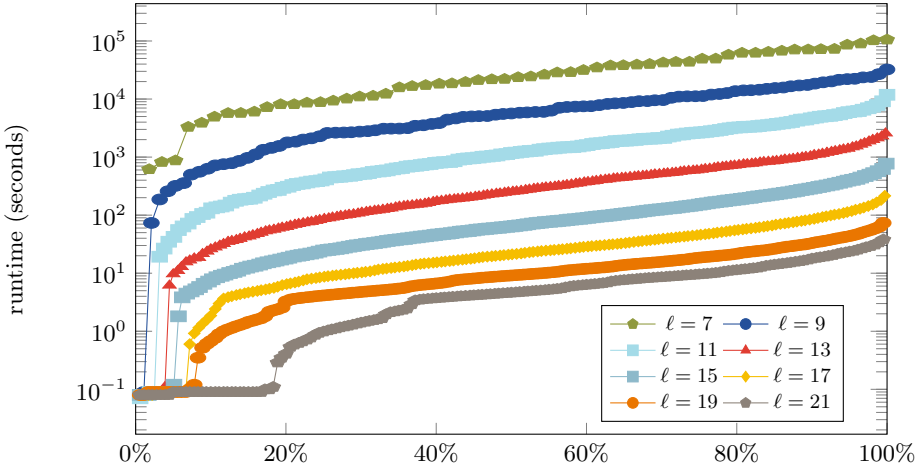


Fig. 6. Runtime to solve the subproblems of Theorem 2 for various splitting parameters.

reduction compared to full partitioning ($\ell = 21$). Note that the solving time for the hardest cube (the max column) increases substantially when using fewer cubes. This in turn reduces the effectiveness of parallelism. The runtime without partitioning is expected to be about 1000 CPU hours, so partitioning achieves super-linear speedups and more than a factor of 4 speedup for $\ell = 15$. Fig. 6 shows plots of cumulatively solved cubes, with similar curves for all settings.

We also evaluated the off-the-shelf tool *March* for partitioning. This tool was used to prove Schur Number Five [16]. We used option `-d 13` to cut off partitioning at depth 13 to create 8192 cubes. That partition turned out to be very poor: at least 18 cubes took over 100 000 seconds. The expected total costs are about 10 000 CPU hours, so 10 times the estimated partition-free runtime.

A partitioning can also guide the search to solve the formula $g(6) \leq 17$. The partitioning of this formula using $\ell = 12$ results in 1108 cubes. If we add these cubes to the formula with the symmetry-predicate (O_4) in the iCNF format [34], then CaDiCaL can solve it in 8.53 seconds using 205 153 conflicts.

Table 2. Runtime comparison for different values of partitioning parameter ℓ

ℓ	#cubes	average time (s)	max time (s)	total time (h)
21	312 418	6.99	66.86	606.55
19	89 384	13.61	123.70	337.96
17	25 663	34.29	293.10	244.50
15	7393	112.61	949.50	231.27
13	2149	431.26	3 347.59	257.44
11	629	1 847.46	11 844.05	322.79
9	188	7 745.14	32 329.05	404.47
7	57	32 905.90	105 937.76	521.01

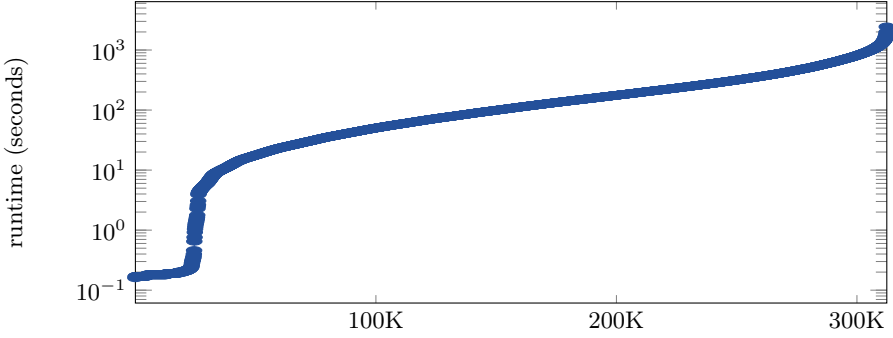


Fig. 7. Reported process time to solve the subproblems of $h(6) \leq 30$ with proof logging while running a formally-verified checker to validate the solver’s output.

6.3 Theorem 1

To show that the optimized encoding for $h(6) \leq 30$ is unsatisfiable, we partitioned the problem with the splitting algorithm described in Section 5 with parameter $\ell = 21$, which results in 312 418 cubes. We picked this setting based on the experiments shown in Table 2. Fig. 7 shows the runtime of solving the subproblems. The average runtime was just below 200 seconds. All subproblems were solved in less than an hour. Almost 24 000 subproblems could be solved within a second. For these subproblems, the cube resulted directly in a conflict, so the solver didn’t have to do any search.

The total runtime is close to 17 300 CPU hours, or slightly less than 2 CPU years. We could achieve practically a linear speedup using 1000 `m6i.xlarge` instances. The timings include producing and validating the proof as described in Section 7.1. The combined size of the proofs is 180 terabytes in the uncompressed LRAT format used by the `cakeLPR` checker. In past verification efforts of hard math problems, the produced proofs were in the DRAT format. For this problem, the LRAT proofs are roughly 2.3 times as large as the corresponding DRAT proof. We estimate that the DRAT proof would have been 78 terabytes in size, so approximately one third to the proof of the Pythagorean triples problem [17]. For all problems, the checker was able to easily catch up with the solver while running on a different core, thereby finishing as soon as the solver was done.

7 Verification

We applied three verification steps to increase trust in the correctness of our results. In the first step, we check the results produced by the SAT solver. The second step consists of checking the correctness of the optimizations discussed in Section 4. In the third step, we validate that the case split covers all cases.

7.1 Concurrent Solving and Checking

The most commonly used approach to validate SAT-solving results works as follows. First, a SAT solver produces a DRAT proof. This proof is checked and trimmed using an unverified efficient tool that produces a LRAT proof. The difference between a DRAT proof and a LRAT proof is that the latter contains hints. The LRAT proof is then validated by a formally-verified checker, which uses the hints to obtain efficient performance.

Recently, the SAT solver **CaDiCaL** added support for producing LRAT proofs directly (since version 1.7.0). This allows us to produce the proof and validate it concurrently. To the best of our knowledge, we are the first to take advantage of this possibility. **CaDiCaL** sends its proof to a pipe and the verified checker **cakeLPR** reads it from the pipe. This tool chain works remarkably well and adds little overhead while avoiding storing large files.

7.2 Reencoding Proof

We validated the four optimizations presented in Section 4. Only the trusted encoding has the reflection symmetry, as each of the optimizations don't preserve this symmetry. Each of the clauses in the symmetry-breaking predicate have the substitution redundancy (SR) property [5] with respect to the trusted encoding. However, there doesn't exist a SR checker. Instead, we transformed the SR check into a sequence of DRAT addition and deletion steps. This is feasible for small point sets (up to 10 points), but is too expensive for the full problem. It may therefore be more practical to verify this optimization in a theorem prover.

Transforming the trusted encoding into the domain-consistent one is challenging to validate because the solver cannot easily infer the existence of a 6-hole using only the clauses (7). Since we are replacing (4) by (7) and clause deletion trivially preserves satisfiability, we only need to check whether each of the clauses (7) is entailed by the trusted encoding. This can be achieved by constructing a formula that asks whether there exists an assignment that satisfies the trusted encoding, but falsifies at least one of the clauses (7). We validated that this formula is unsatisfiable for $n \leq 12$ (around 300 seconds).⁵ The formula becomes challenging to solve for larger n . However, the validation for small n provides substantial evidence of the correctness of the encoding and the implementation.

Checking the correctness of the other two optimizations is easier. Observe that one can obtain the domain-consistent encoding from the $O(n^4)$ encoding by applying Davis-Putnam resolution [6] on the auxiliary variables. This can be expressed using DRAT steps. The DRAT derivation from the domain-consistent encoding to the $O(n^4)$ encoding applies all these steps in reverse order. The minor optimizations mostly delete clauses, which is trivially correct for proofs of unsatisfiability. The clauses (19) have the RAT property on the auxiliary variables and their redundancy can easily be checked using a DRAT checker.

⁵ We implemented an entailment tool, see <https://github.com/marijnheule/entailment>

7.3 Tautology Proof

The final validation step consists of checking whether the partition of the problem covers the entire search space. This part has also been called the tautology proof [16], because in most cases it needs to determine whether the disjunction of cubes is a tautology. We take a slightly different approach and validate that the following formula is unsatisfiable: the conjunction of the negated cubes; the symmetry-breaking predicate; and some clauses from the formula.

Recall that we omitted various cubes because they resulted in a conflict with the clauses $(\overline{o_{a,a+1,a+2}} \vee \overline{o_{a+1,a+2,a+3}} \vee \overline{o_{a+2,a+3,a+4}})$ with $a \in \{2, \dots, n-4\}$ and $(o_{a,a+1,a+2} \vee o_{a+1,a+2,a+3} \vee o_{a+2,a+3,a+4} \vee o_{a+3,a+4,a+5})$ with $a \in \{2, \dots, n-5\}$. We checked with DRATtrim that these clauses are implied by the optimized formulas, which takes 0.3 CPU seconds. We combined them with the negated cubes and the symmetry-breaking predicate, which results in an unsatisfiable formula that can be solved by CaDiCaL in 12 CPU seconds.

8 Conclusion

We closed the final case regarding k -holes in the plane by showing $h(6) = 30$. This is another example that SAT-solving techniques can effectively solve a range of long-standing open problems in mathematics. Other successes include the Pythagorean triples problem [17], Schur number five [16], and Keller’s conjecture [4]. Also, we recomputed $g(6) = 17$ many orders of magnitude faster compared to the original computation by Szekeres and Peters [31] even when taking into account the difference in hardware. So, SAT techniques overwhelmingly outperformed a dedicated approach on this geometry problem. Key contributions include an effective, compact encoding and a partitioning strategy enabling linear-time speedups even when using thousands of cores. We also presented a new concurrent proof-checking procedure to significantly decrease validation costs.

Although the tools are fully automatic, some aspects of our solution require the ingenuity of the user. In particular, we had to develop encoding optimizations and a search-space partitioning strategy to take full advantage of the power of the tools. Constructing the domain-consistent encoding automatically appears challenging. Most other optimizations can be achieved automatically, for example via structured bounded variable elimination [13]. However, the resulting formula cannot be solved as efficiently as the presented one. Substantial research into generating effective partitionings is required to enable non-experts to solve such hard problems. Although we validated most steps, formally verifying the trusted encoding or even the domain-consistent encoding would further increase trust in the correctness of our result.

Acknowledgements Heule is partially supported by NSF grant CCF-2108521. Scheucher was supported by the DFG grant SCHE 2214/1-1. We thank Donald Knuth, Benjamin Kiesel-Reiter, John Mackey, and the reviewers for their valuable feedback. The authors met for the first time during Dagstuhl meeting 23261 “SAT Encodings and Beyond”, which kicked off the research published in this paper.

References

1. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020), <http://hdl.handle.net/10138/318754>
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press, second edn. (2021), <https://www.iospress.com/catalog/books/handbook-of-satisfiability-2>
3. Björner, A., Las Vergnas, M., White, N., Sturmfels, B., Ziegler, G.M.: Oriented Matroids, Encyclopedia of Mathematics and its Applications, vol. 46. Cambridge University Press, 2 edn. (1999). <https://doi.org/10/bhb4rn>
4. Brakensiek, J., Heule, M.J.H., Mackey, J., Narváez, D.E.: The resolution of keller’s conjecture. Journal of Automated Reasoning **66**(3), 277–300 (2022). <https://doi.org/10.1007/S10817-022-09623-5>
5. Buss, S., Thapen, N.: DRAT and propagation redundancy proofs without new variables. Logical Methods in Computer Science **17**(2) (2021). <https://doi.org/10/mbdx>
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7**(3), 201–215 (1960). <https://doi.org/10/bw9h55>
7. Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and Applications of Satisfiability Testing. pp. 502–518. Springer (2004)
8. Erdős, P., Szekeres, G.: A combinatorial problem in geometry. Compositio Mathematica **2**, 463–470 (1935), http://www.renyi.hu/~p_erdos/1935-01.pdf
9. Erdős, P., Szekeres, G.: On some extremum problems in elementary geometry. Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Mathematica **3–4**, 53–63 (1960), https://www.renyi.hu/~p_erdos/1960-09.pdf
10. Felsner, S., Weil, H.: Sweeps, arrangements and signotopes. Discrete Applied Mathematics **109**(1), 67–94 (2001). <https://doi.org/10/dc4tb4>
11. Gent, I.P.: Arc consistency in SAT. In: European Conference on Artificial Intelligence (ECAI 2002). FAIA, vol. 77, pp. 121–125. IOS Press (2002), <https://frontiersinai.com/ecai/ecai2002/pdf/p0121.pdf>
12. Gerken, T.: Empty Convex Hexagons in Planar Point Sets. Discrete & Computational Geometry **39**(1), 239–272 (2008). <https://doi.org/10/c4kn3s>
13. Haberlandt, A., Green, H., Heule, M.J.H.: Effective Auxiliary Variables via Structured Reencoding. In: International Conference on Theory and Applications of Satisfiability Testing (SAT 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 271, pp. 11:1–11:19. Dagstuhl, Dagstuhl, Germany (2023). <https://doi.org/10.4230/LIPIcs.SAT.2023.11>
14. Harborth, H.: Konvexe Fünfecke in ebenen Punktmengen. Elemente der Mathematik **33**, 116–118 (1978), <http://www.digizeitschriften.de/dms/img/?PID=GDZPPN002079801>
15. Heule, M.J.H.: The DRAT format and DRAT-trim checker (2016), [arXiv:1610.06229](https://arxiv.org/abs/1610.06229)
16. Heule, M.J.H.: Schur number five. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence. AAAI’18, AAAI Press (2018)
17. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In: Theory and Applications

- of Satisfiability Testing (SAT 2016). LNCS, vol. 9710, pp. 228–245. Springer (2016). <https://doi.org/10/gkksn>
18. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In: Hardware and Software: Verification and Testing. pp. 50–65. Springer (2012). <https://doi.org/10/f3ss29>
 19. Heule, M.J.H., Scheucher, M.: Happy Ending: An Empty Hexagon in Every Set of 30 Points (Extended Version) (2024), <https://arxiv.org/abs/2403.00737>
 20. Holmsen, A.F., Mojarrad, H.N., Pach, J., Tardos, G.: Two extensions of the Erdős–Szekeres problem. Journal of the European Mathematical Society pp. 3981–3995 (2020). <https://doi.org/10/gsjw4m>
 21. Horton, J.: Sets with no empty convex 7-gons. Canadian Mathematical Bulletin **26**, 482–484 (1983). <https://doi.org/10/chf6dk>
 22. Järvisalo, M., Biere, A., Heule, M.J.H.: Blocked clause elimination. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 129–144. Springer (2010)
 23. Kalbfleisch, J., Kalbfleisch, J., Stanton, R.: A combinatorial problem on convex regions. In: Proc. Louisiana Conf. Combinatorics, Graph Theory and Computing, Congressus Numerantium, vol. 1, Baton Rouge, La.: Louisiana State Univ. pp. 180–188 (1970)
 24. Knuth, D.E.: Axioms and Hulls, LNCS, vol. 606. Springer (1992). <https://doi.org/10/bwfnz9>
 25. Marić, F.: Fast formal proof of the Erdős–Szekeres conjecture for convex polygons with at most 6 points. Journal of Automated Reasoning **62**, 301–329 (2019). <https://doi.org/10/gsjw4r>
 26. Nicolás, M.C.: The Empty Hexagon Theorem. Discrete & Computational Geometry **38**(2), 389–397 (2007). <https://doi.org/10/bw3hnd>
 27. Overmars, M.: Finding Sets of Points without Empty Convex 6-Gons. Discrete & Computational Geometry **29**(1), 153–158 (2002). <https://doi.org/10/cnqmr4>
 28. Scheucher, M.: Two disjoint 5-holes in point sets. Computational Geometry **91**, 101670 (2020). <https://doi.org/10/gsjw2z>
 29. Scheucher, M.: A SAT Attack on Erdős–Szekeres Numbers in \mathbb{R}^d and the Empty Hexagon Theorem. Computing in Geometry and Topology **2**(1), 2:1–2:13 (2023). <https://doi.org/10/gsjw22>
 30. Suk, A.: On the Erdős–Szekeres convex polygon problem. Journal of the AMS **30**, 1047–1053 (2017). <https://doi.org/10/gsjw44>
 31. Szekeres, G., Peters, L.: Computer solution to the 17-point Erdős–Szekeres problem. Australia and New Zealand Industrial and Applied Mathematics **48**(2), 151–164 (2006). <https://doi.org/10/dkb9j3>
 32. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in cakeml. International Journal on Software Tools for Technology **25**(2), 167–184 (2023). <https://doi.org/10/grw7wm>
 33. Tóth, G., Valtr, P.: The Erdős–Szekeres theorem: Upper Bounds and Related Results. In: Combinatorial and Computational Geometry. vol. 52, pp. 557–568. MSRI Publications, Cambridge Univ. Press (2005), <http://www.ams.org/mathscinet-getitem?mr=2178339>
 34. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009. EPTCS, vol. 14, pp. 62–76 (2009). <https://doi.org/10.4204/EPTCS.14.5>

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

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



Synthesis



Fully Generalized Reactivity(1) Synthesis^{*}

Rüdiger Ehlers and Ayrat Khalimov^(✉)

TU Clausthal, Clausthal-Zellerfeld, Germany
{rudiger.ehlers,ayrat.khalimov}@tu-clausthal.de

Abstract. Generalized Reactivity(1) (GR(1)) synthesis is a reactive synthesis approach in which the specification is split into two parts: a symbolic game graph, describing the safe transitions of a system, a liveness specification in a subset of Linear Temporal Logic (LTL) on top of it. Many specifications can naturally be written in this restricted form, and the restriction gives rise to a scalable synthesis procedure – the reasons for the high popularity of the approach. For specifications even slightly beyond GR(1), however, the approach is inapplicable. This necessitates a transition to synthesizers for full LTL specifications, introducing a huge efficiency drop. This paper proposes a synthesis approach that smoothly bridges the efficiency gap from GR(1) to LTL by unifying synthesis for both classes of specifications. The approach leverages a recently introduced canonical representation of omega-regular languages based on a chain of good-for-games co-Büchi automata (COCO). By constructing COCOA for the liveness part of a specification, we can then build a fixpoint formula that can be efficiently evaluated on the symbolic game graph. The COCOA-based synthesis approach outperforms standard approaches and retains the efficiency of GR(1) synthesis for specifications in GR(1) form and those with few non-GR(1) specification parts.

1 Introduction

Reactive synthesis is the process of automatically computing a provably correct reactive system from its formal specification [13]. A safety-critical system is often developed twice: first, when it is described using a formal specification, and second, when a system is implemented according to this specification. The dream of reactive synthesis is to fully eliminate manual implementation phase.

Reactive synthesis is however computationally hard. For specifications in the commonly used linear temporal logic (LTL), checking whether an implementation exists is 2EXPTIME-complete [30]. The classical approach to solve reactive synthesis from LTL is to first translate the LTL formula into a deterministic parity automaton, followed by solving the induced two-player parity game [7]. The system player wins this game if and only if there is an implementation satisfying the specification. It is the first phase of translating LTL to parity automaton that usually represents a bottleneck. This observation spurred a series

^{*} This work has been partially supported by the DFG through Grant No. 322591867 (GUISynth) and the BMWi through Grant No. 19A21026E (SafeWahr).

of synthesis approaches. For instance, in bounded synthesis, either the maximal number of states that a system can have [22] or the longest system response time [20] is restricted. If there exists a system realizing the specification, then there exists one that adheres to some bounds, and bounded synthesis works well whenever small bounds suffice for realizing the given specification. Another approach is to synthesize implementations for parts of the specification, and to then compose them into one that realizes the whole specification [25,31,21]. The approach of [26] avoids constructing one large deterministic parity automaton and instead constructs many smaller ones that—when composed together—represent the original specification. Such decomposition proved beneficial on practical examples [1]. Finally, there are approaches that consider “synthesis-friendly” subsets of LTL. Alur and La Torre identified a number of such LTL fragments with a simpler synthesis problem [3], and this eventually led to the introduction of Generalized Reactivity(1) synthesis by Piterman et al. [28], GR(1) for short. GR(1) synthesis gained a lot of prominence and was applied in domains such as robotics [34,24], cyber-physical system control [36,35], and chip component design [8,23]. We describe it in more detail.

In GR(1) synthesis, the specification is divided into two parts. The first part represents the *safety properties* of a system and encodes a symbolic game graph. Each graph vertex encodes a valuation of last system inputs and outputs. The transitions in the graph represent how these variables can evolve in one step. For instance, a robot on a grid can move from its current cell to the left, right, up, or down, but cannot jump; this is easily encoded as a symbolic game graph. Secondly, there are *liveness properties* of the following form: if certain vertices are visited infinitely often, then certain other vertices must be visited infinitely often as well. The liveness properties are encoded symbolically using LTL formulas of the shape $\bigwedge_i \text{GF} \varphi_i \rightarrow \bigwedge_j \text{GF} \psi_j$, where φ_i and ψ_j are Boolean formulas over input and output system propositions. Synthesis problems from many domains can be encoded naturally, or after some manual effort, into the GR(1) setting.

Constraining specifications to GR(1) form reduces the synthesis problem’s complexity from doubly-exponential to singly-exponential (in the number of propositions), or polynomial when the number of propositions is fixed [8]. The GR(1) synthesis problem can be solved by evaluating a fixpoint formula on the symbolic game arena. The fixpoint formula defines the set of vertices from which the system player satisfies the GR(1) liveness properties while staying in the game arena. The simple shape of GR(1) liveness properties makes the fixpoint formula simple. Moreover, evaluating the fixpoint formula on the symbolic game graph can be done efficiently using Binary Decision Diagrams (BDDs, [12]) as the underlying data structure. These factors together — efficient implementation and relatively expressive specification language — made GR(1) synthesis popular.

GR(1) synthesis has a drawback. A single property outside of GR(1) – for instance, “eventually the robot always stays in some stable zone” (FG *inStableZone*) – makes GR(1) synthesis inapplicable. Switching to full-LTL synthesizers introduces an abrupt efficiency drop, as they do not take advantage of the simple structure of GR(1)-like specifications. For improving the practical applicability

of reactive synthesis, a synthesis approach exhibiting a smooth efficiency curve on the way from GR(1) to LTL would hence be useful. While there are some GR(1) synthesis extensions (e.g., [4,17]), they only extend it by certain specification classes and consequently do not support full LTL.

This paper unifies synthesis for GR(1) and full LTL. Like in GR(1) synthesis, we aim at synthesis for specifications split into the safety part encoded as a symbolic game graph and the liveness part. Unlike the standard GR(1) synthesis, the liveness part can be any LTL or omega-regular property. For standard GR(1) specifications, our approach inherits the efficiency of GR(1) synthesis, including when a specification does not fall syntactically into this class, but is semantically a GR(1) specification. At the same time, for specifications that go beyond GR(1) and only have a few non-GR(1) components, our approach scales well.

Our solution is based on the same fixpoint-evaluation-of-symbolic-game-graph idea. Our starting point is a folklore approach based on solving parity games by evaluating fixpoint equations [11]. We modify it so that it becomes applicable to specifications given in the form of a chain of good-for-games co-Büchi automata (COCOA). Such chains have recently been proposed as a new canonical representation of omega-regular languages [19], and it has been shown how minimal and canonical COCOA can be computed in polynomial time from a deterministic parity automaton of the language. Our COCOA-based synthesis approach converts the liveness part of the specification into a parity automaton, constructs the chain, builds the fixpoint formula from the chain, and finally evaluates it on the symbolic game graph. We show that the fixpoint formula built from the chain has a structure similar to GR(1) fixpoint formulas. This is not the case for the folklore approach via parity games, and as a result, our COCOA-based synthesizer is roughly an order of magnitude faster. The COCOA-based synthesis approach inherits the efficiency of GR(1) synthesis, and it is also efficient on specifications slightly beyond GR(1). Finally, our approach is the first application of the new canonical representation of omega-regular languages.

2 Preliminaries

Automata and languages

Let $\mathbb{N} = \{0, 1, 2, \dots\}$ be the set of natural numbers including 0. Let AP be a set of *atomic propositions*; 2^{AP} denotes the *valuations* of these propositions. A Boolean formula represents a set of valuations: for instance, $\bar{a} \wedge b$, also written $\bar{a}b$, encodes valuations in which proposition a has value *false* and b is *true*. A Boolean function maps valuations of propositions to either *true* or *false*. Binary decision diagrams (BDDs) are a data structure for manipulating such functions.

A *word* is a sequence of proposition valuations $w = x_0x_1 \dots \in (2^{\text{AP}})^{\omega} \cup (2^{\text{AP}})^*$. A word can be finite or infinite. A *language* is a set of infinite words. Given a language L , the *suffix language* of L for some finite word $p \in (2^{\text{AP}})^*$ is $\mathcal{L}(L, p) = \{x_0x_1 \dots \in (2^{\text{AP}})^{\omega} \mid p \cdot x_0x_1 \dots \in L\}$. The words in this set are called *suffix words*. The set of all suffix languages of L is the set $\{\mathcal{L}(L, p) \mid p \in (2^{\text{AP}})^*\}$.

Automata over infinite words are used to finitely represent languages. We consider parity and co-Büchi automata with transition-based acceptance. A *parity automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta)$ with a finite alphabet Σ (usually $\Sigma = 2^{\text{AP}}$), a finite set of states Q , an initial state $q_0 \in Q$, and a finite transition relation $\delta \subseteq Q \times \Sigma \times Q \times \mathbb{N}$ satisfying $(q, x, q', c) \in \delta \Rightarrow (q, x, q', c') \notin \delta$ for all q, x, q' and $c' \neq c$. An automaton is *complete* if for every state q and letter x there exists at least one pair $(q', c) \in Q \times \mathbb{N}$ s.t. $(q, x, q', c) \in \delta$; it is *deterministic* if exactly one such pair (q', c) exists. Wlog. we assume that automata are complete. An automaton is *co-Büchi* if only colors 1 and 2 occur in δ , and then we call the transitions with color 1 *rejecting* and those with color 2 *accepting*.

A *run* of \mathcal{A} on a word $w = x_0x_1 \dots \in \Sigma^\omega$ is a sequence $\pi = \pi_0\pi_1 \dots \in Q^\omega$ starting in $\pi_0 = q_0$ and such that $(\pi_i, x_i, \pi_{i+1}, c_i) \in \delta$ for some c_i for every $i \in \mathbb{N}$; the induced *color sequence* $c = c_0c_1 \dots$ is uniquely defined by w and π . A run is *accepting* if the lowest color occurring infinitely often in the induced color sequence is even (“min-even acceptance”). When this minimal color is uniquely defined, e.g. when there is only one accepting run, it is called *the color of w* wrt. \mathcal{A} . A word is *accepted* if it has an accepting run. The automaton’s language $\mathcal{L}(\mathcal{A})$ is the set of accepted words. The language of the automaton \mathcal{A}' derived from \mathcal{A} by changing the initial state to q is denoted by $\mathcal{L}(\mathcal{A}, q)$.

A *co-Büchi language* is a language representable by a nondeterministic (equiv., deterministic) co-Büchi automaton. The Co-Büchi languages are a strict subset of the omega-regular languages.

An automaton is *good-for-games* if there exists a strategy $f : \Sigma^* \rightarrow Q$ to resolve the nondeterminism to produce accepting runs on the accepted words, formally: for every infinite word $w = x_0x_1 \dots$, the sequence $\pi_0\pi_1 \dots$ defined by $\pi_i = f(x_0 \dots x_{i-1})$ for all $i \in \mathbb{N}$ is a run, *and* it is accepting whenever w belongs to the language.

Games and our realizability problem

LTL. A commonly used formalism to represent system specifications is *Linear Temporal Logic* (LTL, [29]). It uses temporal operators U, X, and derived ones G and F, which we do not define here. For details, we refer the reader to [27].

Games. An edge-labelled *game* is a tuple $G = (\text{AP}_I, \text{AP}_O, V, v_0, \delta, \text{obj})$ where V is a finite set of vertices, $v_0 \in V$ is initial, $\delta : V \times 2^{\text{AP}_I} \times 2^{\text{AP}_O} \rightarrow V$ is a partial function describing possible moves (safety specification), and *obj* is a winning objective (liveness specification). A *play* is a maximal (finite or infinite) sequence of transitions of the form $(v_0, i_0, o_0, v_1)(v_1, i_1, o_1, v_2)(v_2, i_2, o_2, v_3) \dots$; the corresponding sequence $(i_0 \cup o_0)(i_1 \cup o_1) \dots$ is called the *action sequence*. An infinite play is winning for the system if it satisfies the objective *obj*; when *obj* is an LTL objective over $\text{AP}_I \cup \text{AP}_O$, the infinite play satisfies *obj* iff the action sequence satisfies it. A system *strategy* is a function $f : (2^{\text{AP}_I})^+ \rightarrow 2^{\text{AP}_O}$. The game is won by the system if it has a strategy f such that every play $(v_0, i_0, o_0, v_1)(v_1, i_1, o_1, v_2) \dots$ is infinite and it satisfies the objective, where $o_j = f(i_0 \dots i_j)$ for all j . To define parity games, the winning objective *obj* is set to be

a parity-assigning function $obj : V \rightarrow \mathbb{N}$, and then an infinite play satisfies obj iff the minimal parity visited infinitely often in the sequence $obj(v_0)obj(v_1)\dots$ is even (min-even acceptance on states).

The *enforceable predecessor operator* $\square\Diamond$ reads a set of tuples $\Phi \subseteq 2^{\text{AP}} \times V$ and returns the set of positions from which the system can enforce taking one of the transitions into the destination set:

$$\square\Diamond(\Phi) = \{v \in V \mid \forall i. \exists o : (i \cup o, \delta(v, i, o)) \in \Phi\} \quad (1)$$

Symbolic games with LTL objectives. Games can be represented symbolically. For instance, the vertices can be encoded as valuations of Boolean variables AP , and transitions between the vertices can be encoded using a Boolean formula. This paper focuses on solving symbolic games with LTL objectives:

Given a symbolic game with LTL objective. Who wins the game?

The particular symbolic representation is not important as long as it provides the operations for union, intersection, and complementation of sets of label-position tuples, and the enforceable predecessor operator $\square\Diamond$. This paper focuses exclusively on the realizability problem; the extraction of compact and efficient implementations merits a separate study.

Mu-calculus fixpoint formulas. For an introduction to using fixpoint formulas in synthesis, we refer the reader to [7], and to [10,5] for mu-calculus in general. The fixpoint formulas use the greatest (ν) and least (μ) fixpoint operators, and the enforceable-predecessor operator $\square\Diamond$. For instance, the formula $\nu Y. \mu X. \square\Diamond(Y \wedge (\bar{x} \vee X))$ represents the biggest set of vertices such that from all vertices in the set, the system can enforce that either x does not hold along the next transition and this transition leads back to the same set, or the play gets closer to a position from which this can be enforced. This formula hence characterizes the positions from which the system can enforce that \bar{x} holds infinitely often along a play.

Generalized Reactivity(1)

Generalized Reactivity(1) is a class of assume-guarantee specifications that includes safety and liveness components. It gained popularity because many specifications naturally fall into the GR(1) class, and the restricted nature of GR(1) admits an efficient synthesis approach. For the purpose of this paper, we define a GR(1) specification as a game $G_{\text{gr1}} = (\text{AP}_I, \text{AP}_O, V, v_0, \delta, \Phi)$ with an LTL winning objective of the form $\Phi = \bigwedge_{i=1}^m \text{GF}a_i \rightarrow \bigwedge_{j=1}^n \text{GF}g_j$, where each assumption a_i and guarantee g_j are Boolean formulas over $\text{AP}_I \cup \text{AP}_O$. The original GR(1) specification class [28] uses logical formulas to describe the symbolic arena.

Solving GR(1) games using fixpoints

We now show how to solve GR(1) games by evaluating fixpoint formulas on GR(1) game arenas. Consider a GR(1) game $G_{\text{gr1}} = (\text{AP}_I, \text{AP}_O, V, v_0, \delta, \Phi)$ with

$\Phi = \bigwedge_{i=1}^m \text{GF}a_i \rightarrow \bigwedge_{j=1}^n \text{GF}g_j$. The set of positions $W \subseteq V$ from which the system player wins the game is characterized by the fixpoint equation [18,8]:

$$W = \nu Z. \bigwedge_{j=1}^n \mu Y. \bigvee_{i=1}^m \nu X. \Box \Diamond [(g_j \wedge Z) \vee Y \vee (\neg a_i \wedge X)] \quad (2)$$

This fixpoint formula ensures that the system chooses to move into states of one of the three kinds: (1) states where it waits for an environment goal a_i to be reached, possibly forever ($\neg a_i \wedge X$), (2) states that move the system closer to reaching its goal number j (Y), or (3) winning states that satisfy system goal number j ($g_j \wedge Z$). The conjunction over all guarantees to the right of νZ ensures that all liveness guarantees are satisfied from all winning positions (unless some environment liveness assumption is violated). The disjunction over the environment goals permits the system to wait for the satisfaction of any of the environment liveness goals. At the end of evaluating the fixpoint formula, Z consists of the winning positions for the system. The system wins the GR(1) game if and only if W includes v_0 .

Example. Consider a GR(1) game with $\text{AP}_I = \{u\}$, $\text{AP}_O = \{x, y\}$, and $\Phi = \text{GF}u \rightarrow (\text{GF}x \wedge \text{GF}y)$. Equation 2 becomes:

$$W = \nu Z. \left[\begin{array}{l} \mu Y. \nu X. \Box \Diamond (xZ \vee Y \vee \bar{u}X) \wedge \\ \mu Y. \nu X. \Box \Diamond (yZ \vee Y \vee \bar{u}X) \end{array} \right] \quad (3)$$

For conciseness, we write xZ instead of $x \wedge Z$, and \bar{a} instead of $\neg a$.

Solving symbolic parity games using fixpoints

Consider a parity game $(\text{AP}_I, \text{AP}_O, V, v_0, \delta, c)$ with colors $\{0, \dots, n\}$. The winning positions for the system player in such game are characterized by the fixpoint formula from [33,11] adapted to our setting:

$$W = \nu X^0 \mu X^1 \dots \sigma X^n. \Box \Diamond (\bigvee_{i=1}^n \text{color}_i \wedge X^i) \quad (4)$$

The operators ν and μ alternate, so the symbol σ is μ if n is odd and ν if n is even; $\text{color}_i = \{v \mid c(v) = i\}$ denotes the set of vertices of color i .

Solving symbolic LTL games using fixpoints

Let G be a game with LTL objective Φ . We can construct a deterministic parity automaton \mathcal{A} for Φ , build the product parity game $G \otimes \mathcal{A}$, and solve it with the help of Equation 4. An alternative approach is to embed the product into the fixpoint formula by using vector notation [10].

Consider an example. Let $G = (\text{AP}_I, \text{AP}_O, V, v_0, \delta, \Phi)$ be a game with $\Phi = \text{GF}u \rightarrow (\text{GF}x \wedge \text{GF}y)$. The parity automaton for Φ is shown on Figure 1. It has two states, q_0 and q_1 , and uses three colors. For three colors, the parity fixpoint

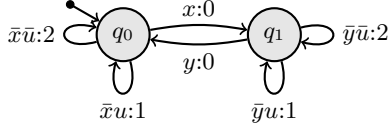


Fig. 1. Parity automaton for $\text{GF}u \rightarrow (\text{GF}x \wedge \text{GF}y)$. Transitions are labeled by the proposition valuations for which they can be taken as well as the color of the transition.

formula in Equation 4 has structure $\nu Z. \mu Y. \nu X$. We index each set variable with the state of the automaton, thus Z is split into Z_0 and Z_1 , etc. The formula is:

$$\begin{bmatrix} W_0 \\ W_1 \end{bmatrix} = \nu \begin{bmatrix} Z_0 \\ Z_1 \end{bmatrix} . \mu \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} . \nu \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} . \Box \Diamond \begin{bmatrix} xZ_1 \vee \bar{x}uY_0 \vee \bar{x}\bar{u}X_0 \\ yZ_0 \vee \bar{y}uY_1 \vee \bar{y}\bar{u}X_1 \end{bmatrix} \quad (5)$$

The top row encodes the transitions from state q_0 of the parity automaton: $q_0 \xrightarrow{x:0} q_1$ becomes xZ_1 , $q_0 \xrightarrow{\bar{x}u:1} q_1$ becomes $\bar{x}uY_1$, $q_0 \xrightarrow{\bar{x}\bar{u}:2} q_0$ becomes $\bar{x}\bar{u}X_0$. After formula evaluation, the variable W_0 contains game positions winning for the system wrt. the parity automaton \mathcal{A}_{q_0} , while W_1 does so wrt. \mathcal{A}_{q_1} .

In general, suppose we are given a game whose winning objective is a deterministic parity automaton $(2^{\text{AP}}, Q, q_0, \delta)$ with transition function $\delta : Q \times \Sigma \rightarrow Q \times \mathbb{N}$ that uses n colors $\{0, \dots, n-1\}$. The set of winning game positions is characterized by the fixpoint formula:

$$\begin{bmatrix} W_1 \\ \vdots \\ W_{|Q|} \end{bmatrix} = \nu \begin{bmatrix} X_1^0 \\ \vdots \\ X_{|Q|}^0 \end{bmatrix} . \mu \begin{bmatrix} X_1^1 \\ \vdots \\ X_{|Q|}^1 \end{bmatrix} \dots \sigma \begin{bmatrix} X_1^{n-1} \\ \vdots \\ X_{|Q|}^{n-1} \end{bmatrix} . \Box \Diamond \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_{|Q|} \end{bmatrix} \quad (6)$$

$$\text{where for all } j \in \{1, \dots, |Q|\}, \text{ we have } \psi_j = \bigvee_{\substack{x \in 2^{\text{AP}} \\ \text{let } (q, c) = \delta(q_j, x)}} x \wedge X_{\text{ind}(q)}^c$$

where $\text{ind} : Q \rightarrow \{1, \dots, |Q|\}$ is some state numbering (one-to-one) that maps the initial automaton state q_0 to 1. The game is won by the system if and only if the initial game position belongs to W_1 .

3 Chains of Good-for-Games co-Büchi Automata

This section reviews the chain of good-for-games co-Büchi automata representation [19] for ω -regular languages used by our synthesis approach in Section 4.

Like parity automata, a chain of co-Büchi automaton representation of a language assigns colors to words. The central difference is that the chain representation relies on a sequence of automata, each taking care of a single color.

Definition 1. Let $L \subseteq \Sigma^\omega$ be an omega-regular language. A falling chain of languages $L_1 \supset L_2 \supset \dots \supset L_n$ is a chain-of-co-Büchi representation of L if

- every language L_i for $i \in \{1, \dots, n\}$ is a co-Büchi language, and

- for every $w \in \Sigma^\omega$, the word w is in L if and only if $w \notin L_1$ or the highest index i such that $w \in L_i$ is even.

Examples. The universal language Σ^ω has the singleton-chain $L_1 = \emptyset$, and the empty language has the chain $(L_1 = \Sigma^\omega) \supset (L_2 = \emptyset)$. The language of the LTL formula $\text{GF}a$ over a single atomic proposition a is expressed by the chain $(L_1 = L(\text{FG}\bar{a})) \supset (L_2 = \emptyset)$, and $L(\text{FG}a)$ by $(L_1 = \Sigma^\omega) \supset (L_2 = L(\text{FG}a)) \supset (L_3 = \emptyset)$.

The definition of the natural color of a word from [19] provides a canonical way to represent L as a chain of co-Büchi languages $L_1 \supset L_2 \supset \dots \supset L_n$, which uses the minimal number of colors. Moreover, Abu Radi and Kupferman describe a procedure to construct a minimal and canonical good-for-games co-Büchi automaton for a given co-Büchi language [2]. Thus, every omega-regular language has a canonical minimal chain-of-co-Büchi-automata representation (**COCOA**).

The canonization procedure in [2, Thm.4.7] ensures the following property.

Lemma 1 ([2]). *Fix a canonical GFG co-Büchi automaton \mathcal{A} computed by [2, Thm.4.7]. For every state q and letter x , either there is*

- exactly one accepting transition, or there are
- one or more rejecting transitions. In this case:
 - all successors of q on x share the same suffix language L' , i.e., for every two successors s_1 and s_2 of q on x : $L(\mathcal{A}, s_1) = L(\mathcal{A}, s_2)$, and
 - for every state q' with suffix language L' , there is a rejecting transition to q' from q on x .

Figure 2 on page 12 shows an example of a COCOA.

Strategies to get back on the track

Every GFG automaton has a strategy to resolve its nondeterminism such that a word is accepted if and only if the run adhering to this strategy is accepting. We allow such strategies to diverge for a finite number of steps, and show that this divergence does not affect the acceptance by canonical GFG automata.

Given a COCOA $\mathcal{A}^1, \dots, \mathcal{A}^n$, define the *natural color* of a word to be the largest level l such that \mathcal{A}^l accepts the word, or 0 if no such l exists. Thus, a word is accepted by the COCOA if and only if the natural color is even.

GFGness strategies f^l . Let $f^l : \Sigma^* \rightarrow Q^l$ be a GFG witness resolving nondeterminism in \mathcal{A}^l , for every $l \in \{1, \dots, n\}$; we call f^l a golden strategy of \mathcal{A}^l , and the induced run for some given word is called its *golden run*.

Restrictions g^l . The synthesis approach, which will be described later, considers combined runs of all automata. Its efficiency depends on the number of reachable states in $Q^1 \times \dots \times Q^n$, so it is beneficial to reduce this number. To this end, we introduce a restriction on successor choices. We first define a helpful notion: for a co-Büchi automaton \mathcal{A} and its state q , let $L^{acc}(q)$ denote the set of words which have a run from q visiting only accepting transitions. For several automata

$\mathcal{A}^1, \dots, \mathcal{A}^l$ and their states q^1, \dots, q^l , define $L^{acc}(q^1, \dots, q^l) = \bigwedge_i L^{acc}(q^i)$. Then, for $l \in \{1, \dots, n\}$, define a *restriction* function $g^l : Q^l \times \Sigma \times Q^1 \times \dots \times Q^{l-1} \rightarrow 2^{Q^l}$: for every $q^l, x, r^1, \dots, r^{l-1}$, let $g^l(q^l, x, r^1, \dots, r^{l-1}) = S \subseteq \delta^l(q^l, x)$ be a maximal set such that for every $r^l \in S$ there exists no other $\tilde{r}^l \in S$ with $L^{acc}(r^1, \dots, r^{l-1}, \tilde{r}^l) \subseteq L^{acc}(r^1, \dots, r^l)$. Intuitively, given a current state q^l of the automaton \mathcal{A}^l , a letter x , and successor states r^1, \dots, r^{l-1} of the automata on lower levels, the function g^l returns a set of states among which \mathcal{A}^l should pick a successor. Runs $\rho^1 = q_0^1 q_1^1 \dots, \dots, \rho^n = q_0^n q_1^n \dots$ of $\mathcal{A}^1, \dots, \mathcal{A}^n$ on a word $x_0 x_1 \dots$ satisfy restrictions g^1, \dots, g^n if for every level $l \in \{1, \dots, n\}$ and step $i \in \mathbb{N}$: $q_{i+1}^l \in g^l(q_i^l, x_i, q_{i+1}^1, \dots, q_{i+1}^{l-1})$. Strategies $f^l : \Sigma^* \rightarrow Q^l$ for $l \in \{1, \dots, n\}$ satisfy restrictions g^1, \dots, g^n if on every word the strategies yield runs satisfying the restrictions.

The following lemma states that requiring runs of $\mathcal{A}^1, \dots, \mathcal{A}^n$ to satisfy the restrictions g^1, \dots, g^n preserves the natural colors and the GFGness.

Lemma 2. *There exist strategies $f^l : \Sigma^* \rightarrow Q^l$ for $l \in \{1, \dots, n\}$ satisfying the restrictions g^1, \dots, g^n such that for every word of a natural color c , the strategies yield accepting runs ρ^1, \dots, ρ^c of $\mathcal{A}^1, \dots, \mathcal{A}^c$.*

Proof. Fix a word w of a natural color c . Each automaton \mathcal{A}^l of the chain has a GFG witness in the form of a strategy $h^l : \Sigma^* \rightarrow Q^l$ to resolve nondeterminism. From such strategies and the restrictions g^1, \dots, g^n , we construct the sought strategies f^1, \dots, f^n , inductively on the level, starting from the smallest level 1 and proceeding upwards to n .

Fix $l \in \{1, \dots, n\}$, and suppose the strategies f^1, \dots, f^{l-1} are already defined; we define the strategy $f^l : \Sigma^* \rightarrow Q^l$. Fix a moment $i - 1$. Let q_{i-1}^l be the state of the run ρ^l proceeding according to f^l , $\tilde{q}_i^l = h^l(x_0 \dots x_{i-1})$ the successor state in the original run $\tilde{\rho}^l$ according to h^l , q_i^1, \dots, q_i^{l-1} the successor states in $\rho^1, \dots, \rho^{l-1}$ adhering to f^1, \dots, f^{l-1} , and $Q_i^l = g^l(q_{i-1}^l, x_{i-1}, q_i^1, \dots, q_i^{l-1})$ the allowed successors on level l . Then:

- if $Q_i^l = \{q_i^l\}$ describes a unique choice, then $f^l(x_0 \dots x_{i-1}) = q_i^l$ takes it,
- else f^l picks any $q_i^l \in Q_i^l$ s.t. $L^{acc}(q_i^1, \dots, q_i^{l-1}, q_i^l) \supseteq L^{acc}(q_i^1, \dots, q_i^{l-1}, \tilde{q}_i^l)$. Note that such q_i^l always exists because in canonical GFG co-Büchi automata a choice of a nondeterministic transition does not narrow the subsequent nondeterminism resolution.

We now show that the strategies f^1, \dots, f^l preserve the natural colors. Fix a word w . It suffices to prove that the original strategy h^l yields an accepting run $\tilde{\rho}^l$ if and only if f^l yields an accepting run ρ^l . If $\tilde{\rho}^l$ is rejecting, then ρ^l is also rejecting, for h^l is a witness of GFGness. Now assume that $\tilde{\rho}^l$ is accepting. After some moment m , the runs $\rho^1, \dots, \rho^{l-1}, \tilde{\rho}^l$ never make a rejecting transition, hence $w_m w_{m+1} \dots \in L^{acc}(q_m^1, \dots, q_m^{l-1}, \tilde{q}_m^l)$. Let $m' \geq m$ be the first moment after m when ρ^l visits a rejecting transition; if no such m' exists, we are done. At moment m' , the strategy f^l picks a successor $q_{m'+1}^l$ such that $L^{acc}(q_{m'+1}^1, \dots, q_{m'+1}^l) \supseteq L^{acc}(q_{m'+1}^1, \dots, \tilde{q}_{m'+1}^l)$. Since $w_{m'+1} \dots \in L^{acc}(q_{m'+1}^1, \dots, q_{m'+1}^{l-1}, \tilde{q}_{m'+1}^l)$, that suffix also belongs to a larger L^{acc} wrt. $q_{m'+1}^l$. Hence the run ρ^l is accepting. \square

Get-back strategies f_\star^l . We now consider runs that diverge from golden runs. Given an individual strategy $f^l : \Sigma^* \rightarrow Q^l$, define $f_\star^l : \Sigma^* \times Q^l \times \Sigma \rightarrow Q^l$ to be a strategy-like function which, when presented with a choice, makes the same choice as f^l . Formally: for every $p \in \Sigma^*$, $q \in Q^l$ reachable from the initial state on reading p , and $x \in \Sigma$, the value $f_\star^l(p, q, x) = f^l(p \cdot x)$ if \mathcal{A}^l needs to take a rejecting transition from q on x , otherwise there is no choice to be made and $f_\star^l(p, q, x) = q'$ for the unique successor q' of q on reading x . It follows from properties of canonical GFG automata (Lemma 1) that every successor chosen by f_\star^l satisfies the transition relation of \mathcal{A}^l . We now prove that it is sufficient to adhere to f_\star^l only eventually.

Lemma 3. *Fix a COCOA and a word w . For $l \in \{1, \dots, n\}$, suppose \mathcal{A}^l on w has a rejecting run ρ^l that eventually adheres to f_\star^l , where f_\star^l is constructed from f^l of Lemma 2. Then \mathcal{A}^l rejects w .*

The proof is based on Lemma 1, which implies that two diverging runs of a canonical GFG automaton on the same word can always be converged once a rejecting transition is taken.

Proof. For $l = 0$ the claim trivially holds; assume $l > 0$. Let ρ_\star^l be the golden run of \mathcal{A}^l on the word. Let m be the moment starting from which ρ^l adheres to the golden strategy of \mathcal{A}^l . Let n be the first moment $n \geq m$ when \mathcal{A}^l makes a rejecting transition: by properties of canonical GFG automata (Lemma 1), there must be a rejecting transition to the same state as in ρ_\star^l . The strategy f_\star^l moves the automaton \mathcal{A}^l in ρ^l into the same state at moment $n + 1$ as it is in ρ_\star^l . Afterwards, the strategy f_\star^l ensures that \mathcal{A}^l in ρ^l follows exactly the same transitions as \mathcal{A}^l in ρ_\star^l . Hence, the golden run ρ_\star^l is rejecting: \mathcal{A}^l rejects w . \square

COCOA product

In this section, we compose individual automata of COCOA into a product which is a good-for-games alternating parity automaton [9]. The results above imply that the languages of a COCOA and its product coincide. Later we use COCOA products to solve games with LTL objectives.

Alternating automata. A simple¹ alternating parity automaton (Σ, Q, q_0, δ) has a transition function of type $\delta : Q \times \Sigma \rightarrow 2^Q \times \mathbb{N} \times \{rej, acc\}$. For instance, $\delta(q, x) = (\{q_1, q_2\}, 1, rej)$ means that from state q on reading letter x there are transitions to q_1 and q_2 , both labelled with color 1, and the choice between q_1 and q_2 is controlled by the rejector player. There are two players, rejector and acceptor, and the acceptance of a word $w = x_0x_1\dots$ is defined via the following word-checking game. Starting in q_0 , the two players resolve nondeterminism and build a play $(q_0, c_0, pl_0, q_1)(q_1, c_1, pl_1, q_2)\dots$: suppose the play sequence is in state

¹ ‘Simple’ refers to a simpler form of the transition function. We use $\delta : Q \times \Sigma \rightarrow 2^Q \times \mathbb{N} \times \{rej, acc\}$ while the general form is $\delta : Q \times \Sigma \rightarrow \mathbf{B}^+(Q)$ plus parity assignment $Q \times \Sigma \times Q \rightarrow \mathbb{N}$. We forbid mixing conjunctions and disjunctions.

q_i , let $\delta(q_i, x_i) = (Q_{i+1}, c_i, pl_i)$: if $pl_i = rej$ then the rejector chooses a state $q_{i+1} \in Q_{i+1}$, otherwise the acceptor chooses. The play sequence is then extended by $(q_i, c_i, pl_i, q_{i+1})$ and the procedure repeats from state q_{i+1} . The play is *won* by the acceptor if the minimal color appearing infinitely often in $c_0 c_1 \dots$ is even (min-even acceptance), otherwise it is won by the rejector. The word-checking game is *won* by the acceptor if it has a strategy $f_w : Q^* \rightarrow Q$ to resolve its nondeterminism to win every play; otherwise the game is won by the rejector, who then also has a winning strategy. Note that although the acceptor strategy does not know the rejector choices beforehand, it knows the word w . The word is *accepted* by the automaton if the word-checking game is won by the acceptor.

A simple alternating automaton is *good-for-games*, abbreviated *A-GFG*, if the acceptor player has a strategy $f_{acc} : (Q \times \Sigma)^* \rightarrow Q$ to win the word-checking game for every accepting word, and the rejector player has a strategy $f_{rej} : (Q \times \Sigma)^* \rightarrow Q$ winning for every rejected word. These strategies depend only on the currently seen word prefix, not the whole word. We remark that our definition of GFGness differs from [9] but they show the equivalence [9, Thm.8].

COCOA product. The product is built in three steps. First, we define a naive product, which combines individual chain automata into A-GFG in a straightforward way. The naive product may contain states whose removal does not affect its language, hence in the second step we define a product with reduced sets of states and transitions. In turn, the reduced product may miss transitions beneficial for synthesis. Therefore, in the last step, we enrich the reduced product with transitions to derive the optimized, and final, COCOA product.

Given a COCOA $\mathcal{A}^l = (\Sigma, Q^l, q_0^l, \delta^l)$ with $l \in \{1, \dots, n\}$, the *naive COCOA product* is the following simple alternating parity automaton (Σ, Q, q^0, δ) . Each state is a tuple from $Q^1 \times \dots \times Q^n$, $q^0 = (q_0^1, \dots, q_0^n)$, and the set of states consists of those reachable from the initial state under the transition relation defined next. The transition relation $\delta : Q \times \Sigma \rightarrow 2^Q \times \mathbb{N} \times \{rej, acc\}$ simulates individual automata of the COCOA. Consider an arbitrary $(q^1, \dots, q^n) \in Q$, $x \in \Sigma$; let r be the smallest number such that \mathcal{A}^r has a rejecting transition from q^r on reading x , i.e., $(q^r, x, \tilde{q}^r, 1) \in \delta^r$ for some $\tilde{q}^r \in Q^r$, otherwise set r to $n+1$. By abuse of notation, define $\delta^l(q^l, x) = \{\tilde{q}^l \mid \exists p : (q^l, x, \tilde{q}^l, p) \in \delta^l\}$ to be the set of successor states of q^l on reading x in \mathcal{A}^l . Let pl^r be *rej* for odd r and *acc* for even r . Then, $\delta((q^1, \dots, q^n), x) = (\tilde{Q}, r-1, pl^r)$, where:

$$\tilde{Q} = \{(\tilde{q}^1, \dots, \tilde{q}^n) \mid \tilde{q}^l \in \delta^l(q^l, x) \text{ for every } l\}.$$

Notice that the automata on levels $l < r$ have unique successors (\tilde{q}^l is unique) as their transitions are accepting and hence deterministic (by Lemma 1 on page 8). The automata on levels $l \geq r$ may need to resolve nondeterminism, which is done by a single player pl^r in the product.

The *reduced COCOA product* is defined by replacing the definition of \tilde{Q} by

$$\tilde{Q} = \{(\tilde{q}^1, \dots, \tilde{q}^n) \mid \tilde{q}^l \in g^l(\tilde{q}^1, \dots, \tilde{q}^{l-1}, x, q^l) \text{ for every } l\}$$

where the restriction function g^l was defined on page 9. As a result, this set \tilde{Q} has no two states (q^1, \dots, q^n) and $(\tilde{q}^1, \dots, \tilde{q}^n)$ with $L^{acc}(q^1, \dots, q^n) \subseteq L^{acc}(\tilde{q}^1, \dots,$

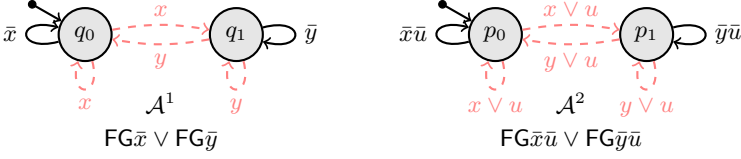


Fig. 2. COCOA for the language $GFu \rightarrow (GFx \wedge GFy)$. Rejecting transitions are dashed.

\tilde{q}^n). The set of states of the reduced COCOA product is the set of states from $Q^1 \times \dots \times Q^n$ reachable under the above definition.

Finally, given a reduced COCOA product $(\Sigma, Q, q^0, \delta_R)$, we now define the *optimized COCOA product* $(\Sigma, Q, q^0, \delta_O)$. It has the same states Q as the reduced product but adds transitions. For $(q^1, \dots, q^n) \in Q$, $x \in \Sigma$, let $(\tilde{Q}_R, r-1, pl^r) = \delta_R((q^1, \dots, q^n), x)$. Then $\delta_O((q^1, \dots, q^n), x) = (\tilde{Q}, r-1, pl^r)$, where

$$\begin{aligned} \tilde{Q} &= \tilde{Q}_R \cup \{(\tilde{q}^1, \dots, \tilde{q}^n) \in Q : \\ &\quad \forall l \in \{1, \dots, r-1\}: q^l \in \delta^l(q^l, x) \wedge \\ &\quad \forall l \in \{r, \dots, n\}. \exists (\tilde{q}_R^1, \dots, \tilde{q}_R^n) \in \tilde{Q}_R: L(\tilde{q}^l) = L(\tilde{q}_R^l)\}. \end{aligned}$$

In the first condition, the successor q^l for $l \leq r-1$ is uniquely defined. The second condition on levels higher than $r-1$ allows for state jumping.

Lemma 4. *For every COCOA, the optimized product is A-GFG and has the same language as the COCOA.*

Proof. We describe two strategies, $f_{\text{acc}} : (Q \times \Sigma)^* \rightarrow Q$ for the acceptor and $f_{\text{rej}} : (Q \times \Sigma)^* \rightarrow Q$ for the rejector, and prove two claims: for every word, (1) if the word is accepted by COCOA, the acceptor wins the word-checking game using f_{acc} , (2) if the word is rejected by COCOA, the rejector wins the word-checking game using f_{rej} . The lemma follows from these claims.

We define f_{acc} . Given a finite history $h = ((q_1^1, \dots, q_1^n), x_1) \dots ((q_i^1, \dots, q_i^n), x_i)$, let $f_{\text{acc}}(h) = (q_{i+1}^1, \dots, q_{i+1}^n)$, where for $l = 1, \dots, n$:

- if l is even: $q_{i+1}^l = f_{\star}^l(x_1 \dots x_{i-1}, q_i^l, x_i)$;
- if l is odd, pick arbitrary $q_{i+1}^l \in g^l(q_{i+1}^1, \dots, q_{i+1}^{l-1}, q_i^l)$.

The strategy f_{rej} is built similarly but f_{\star}^l is used for odd l . Finally, the two items are then proven using contraposition and then applying Lemma 3. \square

Example. Figure 3 shows the optimized product for COCOA in Figure 2.

4 Solving LTL Games Using Chain of co-Büchi Automata

This section shows how to solve symbolic games with LTL objectives by going through COCOA. For a given LTL specification we construct a deterministic

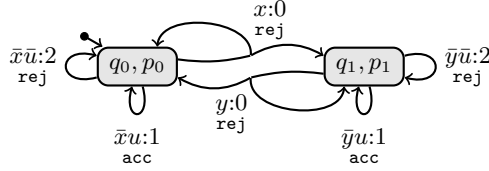


Fig. 3. Optimized COCOA product for $\text{GF}u \rightarrow (\text{GF}x \wedge \text{GF}y)$. It has only two nondeterministic transitions, connecting (q_0, p_0) and (q_1, p_1) , controlled by the rejector. For instance, $\delta((q_0, p_0), x) = (\{(q_0, p_0), (q_1, p_1)\}, 0, \text{rej})$.

parity automaton and then a COCOA using the effective procedure of [19]. We then compute the COCOA product. Finally, we encode the symbolic game with a COCOA product objective into a fixpoint formula. The latter step is simple because the COCOA product is a good-for-games alternating automaton, and such automata are composable with games [9, Thm.8]. Finally, we show that the GR(1) fixpoint equation is a special case of the COCOA fixpoint formula.

Fixpoint formula for games with COCOA objectives

Given a game with an objective in the form of an optimized COCOA product $(2^{\text{AP}}, Q, q_0, \delta)$, we construct a fixpoint formula that characterizes the set of winning positions. Since the COCOA product is a good-for-games parity automaton, the formula resembles Equation 6. It has the structure $\nu X^0. \mu X^1. \dots \sigma X^n$ where $n + 1$ is the number of colors in the COCOA product, and the operators ν and μ alternate. As before, we use the vector notation, and split each variable X^l into $|Q|$ variables $X_1^l, \dots, X_{|Q|}^l$, one per state of the COCOA product, and the k th row in the fixpoint formula encodes transitions from state q_k of the product. Let $\text{ind} : Q \rightarrow \{1, \dots, |Q|\}$ be some one-to-one state numbering with the initial state of the COCOA product mapped to 1, and let OP^{pl} denote \bigvee when pl is *acc* otherwise it is \bigwedge . The following fixpoint formula computes, for each state q of the COCOA product, the set $W_{\text{ind}(q)}$ of game positions from which the system player wins the game wrt. the COCOA product whose initial state is set to q :

$$\begin{aligned} \begin{bmatrix} W_1 \\ \vdots \\ W_{|Q|} \end{bmatrix} &= \nu \begin{bmatrix} X_1^0 \\ \vdots \\ X_{|Q|}^0 \end{bmatrix} \cdot \mu \begin{bmatrix} X_1^1 \\ \vdots \\ X_{|Q|}^1 \end{bmatrix} \dots \sigma \begin{bmatrix} X_1^n \\ \vdots \\ X_{|Q|}^n \end{bmatrix} \cdot \Box \Diamond \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_{|Q|} \end{bmatrix}, \text{ where for all } j: \quad (7) \\ \psi_j &= \bigvee_{\substack{x \in 2^{\text{AP}} \\ \text{let } (\bar{Q}, c, pl) = \delta(q_j, x)}} \left(x \wedge \text{OP}^{pl}_{q \in \bar{Q}} X_{\text{ind}(q)}^c \right) \end{aligned}$$

The game wrt. the COCOA product is won by the system player if and only if $v_0 \in W_1$. Since the languages of COCOA and its optimized product coincide (Lemma 4), we arrive at the following theorem.

Theorem 1. *A game with an LTL objective Φ is won by the system if and only if the initial game position belongs to W_1 computed by Equation 7 for the optimized COCOA product for Φ .*

Example. Consider the LTL specification $\text{GF}u \rightarrow (\text{GF}x \wedge \text{GF}y)$. The optimized product contains only states (q_0, p_0) and (q_1, p_1) . The fixpoint formula is:

$$\nu \begin{bmatrix} Z_{00} \\ Z_{11} \end{bmatrix} \cdot \mu \begin{bmatrix} Y_{00} \\ Y_{11} \end{bmatrix} \cdot \nu \begin{bmatrix} X_{00} \\ X_{11} \end{bmatrix} \cdot \Box \Diamond \begin{bmatrix} xZ_{00}Z_{11} \vee \bar{x}uY_{00} \vee \bar{x}\bar{u}X_{00} \\ yZ_{00}Z_{11} \vee \bar{y}uY_{11} \vee \bar{y}\bar{u}X_{11} \end{bmatrix}$$

where the subscript index ij denotes a state (q_i, p_j) of the optimized COCOA product. The LTL game is won by the system if and only if at the end of evaluation the initial game position v_0 belongs to Z_{00} . This formula has a structure similar to the GR(1) Equation 3, in particular it uses the conjunction over Z variables which leads to a reduction of the number of fixpoint iterations. In contrast, the parity formula in Equation 5 misses this acceleration.

GR(1) synthesis as a special case

We argue that for GR(1) specifications, the COCOA fixpoint Equation 7 becomes similar – in spirit – to GR(1) fixpoint Equation 2. Consider a GR(1) formula $\bigwedge_{i=1}^m \text{GF}a_i \rightarrow \bigwedge_{j=1}^n \text{GF}g_j$. Its COCOA has two automata, \mathcal{A}^1 and \mathcal{A}^2 . The automaton \mathcal{A}^1 accepts exactly the words that violate one of the guarantees, while \mathcal{A}^2 accepts exactly the words that violate one of the guarantees and one of the assumptions. In order to reason about a number of states in canonical automata, we assume henceforth that in the GR(1) formula, no assumption implies another assumption or guarantee, and no guarantee implies another guarantee. The structures of \mathcal{A}^1 and \mathcal{A}^2 are as follows. The automaton \mathcal{A}^1 has one state per guarantee (n in total), while \mathcal{A}^2 has one per combination of liveness assumption and guarantee ($m \cdot n$ in total). The optimized COCOA product has exactly one state for each assumption-guarantee combination, $m \cdot n$ in total, versus $n \cdot (m \cdot n)$ for the non-optimized product. Let $\{1, \dots, m\} \times \{1, \dots, n\}$ be the states of the optimized product, and let $(1, 1)$ be initial. For each state (i, j) :

- for every $x \models \bar{a}_i \bar{g}_j$: $\delta((i, j), x) = (\{(i, j)\}, 2, \text{rej})$,
- for every $x \models a_i \bar{g}_j$: $\delta((i, j), x) = (\{(i', j) \mid i' \in \{1, \dots, m\}\}, 1, \text{acc})$, and
- for every $x \models g_j$: $\delta((i, j), x) = (\{1, \dots, m\} \times \{1, \dots, n\}, 0, \text{rej})$.

The fixpoint formula for such COCOA product has the form:

$$\begin{bmatrix} W_{1,1} \\ \vdots \\ W_{m,n} \end{bmatrix} = \nu \begin{bmatrix} Z_{1,1} \\ \vdots \\ Z_{m,n} \end{bmatrix} \cdot \mu \begin{bmatrix} Y_{1,1} \\ \vdots \\ Y_{m,n} \end{bmatrix} \cdot \nu \begin{bmatrix} X_{1,1} \\ \vdots \\ X_{m,n} \end{bmatrix} \cdot \Box \Diamond \begin{bmatrix} \psi_{1,1} \\ \vdots \\ \psi_{m,n} \end{bmatrix}, \text{ where for all } i, j:$$

$$\psi_{i,j} = g_j \left(\bigwedge_{\substack{i' \in \{1, \dots, m\} \\ j' \in \{1, \dots, n\}}} Z_{i',j'} \right) \vee a_i \bar{g}_j \left(\bigvee_{i' \in \{1, \dots, m\}} Y_{i',j} \right) \vee \bar{a}_i \bar{g}_j X_{i,j}$$

The conjunction $\bigwedge_{i',j'} Z_{i',j'}$ and disjunctions $\bigvee_{i'} Y_{i',j}$ enable faster information propagation which results in smaller number of fixpoint iterations. Such information sharing is present in GR(1) fixpoint Equation 2, and it is in this sense the COCOA approach generalizes GR(1) approach. In contrast, the parity fixpoint formula for GR(1) specifications misses this acceleration.

We now optimize the equation to reduce the number of variables. First, we introduce variables Y_j and Z_j , for $j \in \{1, \dots, n\}$, and transform the formula into

$$\begin{aligned} \begin{bmatrix} W_1 \\ \vdots \\ W_n \end{bmatrix} &= \nu \begin{bmatrix} Z_1 \\ \vdots \\ Z_n \end{bmatrix} \cdot \mu \begin{bmatrix} Y_1 \\ \vdots \\ Y_n \end{bmatrix} \cdot \begin{bmatrix} \bigvee_i \Phi_{i,1} \\ \vdots \\ \bigvee_i \Phi_{i,n} \end{bmatrix}, \text{ where} \\ \begin{bmatrix} \Phi_{1,1} \\ \vdots \\ \Phi_{m,n} \end{bmatrix} &= \nu \begin{bmatrix} X_{1,1} \\ \vdots \\ X_{m,n} \end{bmatrix} \cdot \square \diamond \begin{bmatrix} \psi_{1,1} \\ \vdots \\ \psi_{m,n} \end{bmatrix}, \text{ where} \\ \psi_{i,j} &= g_j \left(\bigwedge_{j' \in \{1, \dots, n\}} Z_{j'} \right) \vee a_i \bar{g}_j Y_j \vee \bar{a}_i \bar{g}_j X_{i,j} \end{aligned}$$

Note that for every $i \in \{1, \dots, m\}$, the value $W_{i,j}$ computed by the old formula equals the value W_j computed by the new formula ($W_{i,j} = W_i$), where $j \in \{1, \dots, n\}$. We then introduce a fresh variable Z , and transform the formula to:

$$\begin{aligned} W &= \nu Z. \bigwedge_{j \in \{1, \dots, n\}} \Psi_j, \text{ where} \\ \begin{bmatrix} \Psi_1 \\ \vdots \\ \Psi_n \end{bmatrix} &= \mu \begin{bmatrix} Y_1 \\ \vdots \\ Y_n \end{bmatrix} \cdot \begin{bmatrix} \bigvee_i \Phi_{i,1} \\ \vdots \\ \bigvee_i \Phi_{i,n} \end{bmatrix}, \text{ where} \\ \begin{bmatrix} \Phi_{1,1} \\ \vdots \\ \Phi_{m,n} \end{bmatrix} &= \nu \begin{bmatrix} X_{1,1} \\ \vdots \\ X_{m,n} \end{bmatrix} \cdot \square \diamond \begin{bmatrix} g_1 Z \vee a_1 \bar{g}_1 Y_1 \vee \bar{a}_1 \bar{g}_1 X_{1,1} \\ \vdots \\ g_n Z \vee a_n \bar{g}_n Y_n \vee \bar{a}_n \bar{g}_n X_{m,n} \end{bmatrix} \end{aligned}$$

After this transformation, we have $W = W_j$ for every $j \in \{1, \dots, n\}$. Finally, the last equations can be folded into the formula

$$W = \nu Z. \bigwedge_{j=1}^n \mu Y. \bigvee_{i=1}^m \nu X. \square \diamond [g_j Z \vee a_i \bar{g}_j Y \vee \bar{a}_i \bar{g}_j X]$$

which is equal to Equation 2 modulo expressions in front of the variables. Our prototype tool implements a generalized version of such formula optimization.

5 Evaluation

Evaluation goals are: (G1) show that standard LTL synthesizers do not fit our synthesis problem, (G2) compare our approach against specialized GR(1) synthesizer, and (G3) compare the COCOA approach against the parity approach.

We implemented COCOA and parity approaches in a prototype tool **reboot**. It uses SPOT [16] to convert LTL specifications (the liveness part of GR(1)) to deterministic parity automata. From it, **reboot** builds COCOA using the construction described in [19]. The COCOA is then compiled into a fixpoint formula in Equation 7, and symbolically evaluated on the game graph. For symbolic encoding of game positions and transitions, we use the BDD library CUDD [32].

We compare our approaches with GR(1) synthesis tool **slugs** [18] and the LTL synthesis tool **strix** [26] which represent the state of the art. The experiments were performed on a Linux machine with AMD EPYC 7502 processor; the timeout was set to 1 hour. To implement the comparison, we collected existing and created new benchmarks: AMBA, lift, and robot on a grid. Each specification is written in an extension of the **slugs** format: it encodes a symbolic game graph using logical formulas over system and environment propositions, and an LTL property on top of it. In total, there are 80 benchmarks, all realizable.

The evaluation data is available at <https://doi.org/10.5281/zenodo.10448487>

AMBA and lift. We use two parameterized benchmarks inspired by [8], each having two versions, a GR(1) and an LTL version. The first specification encodes an elevator behaviour and is parameterized by the number of floors. Its GR(1) specification has one liveness assumption and a parameterized number of guarantees ($GF \rightarrow \bigwedge_i GF$). Lift’s LTL version adds an additional request-response assumption and has the form $GF \wedge (GF \rightarrow GF) \rightarrow \bigwedge_i GF$, which requires 5 parity colors. There are 24 GR(1) instances and 21 LTL instances, with the number of Boolean propositions ranging from 7 to 34. The AMBA specification describes the behaviour of an industrial on-chip bus arbiter serving a parameterized number of clients. Its GR(1) version has the shape $GF \rightarrow \bigwedge_i GF$; our new LTL modification replaces one safety guarantee φ by $FG\varphi$, which allows the system to violate it during some initial phase, and we add an assumption of the form $GF \rightarrow GF$. Overall, the AMBA’s LTL specification has the form $GF \wedge (GF \rightarrow GF) \rightarrow FG \wedge \bigwedge_i GF$, and requires 7 parity colors. There are 14 GR(1) instances and 7 LTL instances; the number of Boolean propositions is 22 for the specification serving two clients, and 77 for the 15-client version.

Robot on a grid. This benchmark describes the standard scenario from robotics domain: a robot moves on a grid, there are walls, doors, pickup and delivery locations, and a moving obstacle. When requested, the robot has to pickup a package and deliver it to the target location, while avoiding collisions with the walls and the obstacle and passing through the doors only when they are open. The GR(1) specification has parameterized number of assumptions and guarantees: $\bigwedge_i GF \rightarrow \bigwedge_i GF$. The LTL version introduces preferential paths: the robot has to eventually always use it assuming that the moving obstacle only moves along her preferred path. This yields the shape $FG \wedge \bigwedge_i GF \rightarrow FG \wedge \bigwedge_i GF$ (5 colors). There are 16 maps of size 8×16 with varying number of delivery-pickup locations and doors. The number of Boolean propositions ranges from 24 to 53.

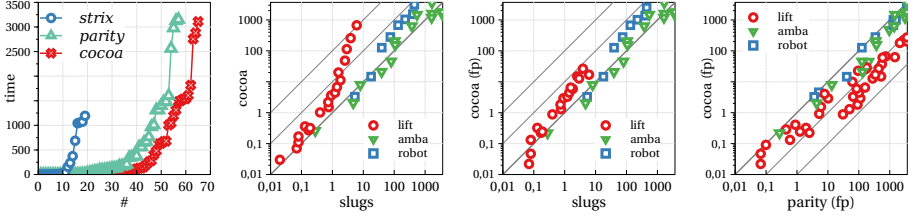


Fig. 4. From left to right: (G1) Cactus plot comparing our approaches with LTL synthesizer *strix* [26]; (G2a) Comparing COCOA-based approach with GR(1) synthesizer *slugs* [17]; (G2b) The same but excluding LTL-to-parity translation time; (G3) Comparing COCOA and parity approaches (excluding LTL-to-parity translation time).

G1: Comparing with LTL synthesizer. Figure 4 shows a cactus plot. On these problems, the LTL synthesizer *strix* is slower than specialized solvers. The reason is the sheer number of states in benchmark game arenas: e.g., benchmark *amba15* uses 77 Boolean propositions, yielding the naive estimate of game arena size in 2^{77} states. Solver *strix* tries to construct an explicit-state automaton describing this game arena and the LTL property, which is a bottleneck. In contrast, symbolic solvers like *slugs* or *reboot* represent game arenas symbolically using BDDs, and *reboot* constructs explicit automata only for LTL properties.

G2: Comparing with GR(1) synthesizer. The second diagram in Figure 4 compares the COCOA approach with *slugs* on the GR(1) benchmarks. The diagram shows the total solving time, including the time *reboot* spends calling SPOT for translating GR(1) liveness formula to parity automaton. On Lift examples, most of the time is spent in this translation when the number of floors exceeds 15: for instance, on benchmark *lift20* *reboot* spent 650 out of total 670 seconds in translation. If we count only the time spent in fixpoint evaluation – and that is a more appropriate measure since GR(1) liveness formulas have a fixed structure – the performances are comparable, see the third diagram.

G3: COCOA vs. parity. The last diagram in Figure 4 compares COCOA and parity approaches on all the benchmarks, and shows that the COCOA approach is significantly faster than the parity one. We note that on these examples, the number of states in the optimized COCOA product was equal to or less than the number of states in the parity automaton. At the same time, the number of fixpoint iterations performed by the COCOA approach was always significantly smaller than for the parity one. Intuitively, this is due to the structure of COCOA fixpoint equation that propagates information faster than the parity one.

Remarks. We did not compare with other symbolic approaches for solving parity or Rabin games [15,14,6]: although they use symbolic algorithms, as input these tools require games in explicit form or their game encoding separates positions into those of player-1 and player-2; both significantly affects the performance.

While all our benchmarks were realizable, the prototype tool was systematically compared against other approaches on both realizable and unrealizable random specifications using fuzz testing.

References

1. Reactive synthesis competition SyntComp 2023: Results. <http://www.syntcomp.org/syntcomp-2023-results>, accessed: 15-09-2023
2. Abu Radi, B., Kupferman, O.: Minimization and canonization of GFG transition-based automata. *Log. Methods Comput. Sci.* **18**(3) (2022)
3. Alur, R., Torre, S.L.: Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.* **5**(1), 1–25 (2004)
4. Amram, G., Maoz, S., Pistiner, O.: GR(1)*: GR(1) specifications extended with existential guarantees. In: *Third World Congress on Formal Methods (FM)*. pp. 83–100 (2019)
5. Arnold, A., Niwiński, D.: *Rudiments of mu-calculus*. Elsevier (2001)
6. Banerjee, T., Majumdar, R., Mallik, K., Schmuck, A.K., Soudjani, S.: Fast symbolic algorithms for omega-regular games under strong transition fairness. *TheoretiCS* **2** (2023)
7. Bloem, R., Chatterjee, K., Jobstmann, B.: *Graph Games and Reactive Synthesis*, pp. 921–962. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_27, https://doi.org/10.1007/978-3-319-10575-8_27
8. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa’ar, Y.: Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.* **78**(3), 911–938 (2012)
9. Boker, U., Lehtinen, K.: Good for Games Automata: From Nondeterminism to Alternation. In: Fokink, W., van Glabbeek, R. (eds.) *30th International Conference on Concurrency Theory (CONCUR 2019)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 140, pp. 19:1–19:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). <https://doi.org/10.4230/LIPIcs.CONCUR.2019.19>, <http://drops.dagstuhl.de/opus/volltexte/2019/10921>
10. Bradfield, J.C., Walukiewicz, I.: The mu-calculus and model checking. In: *Handbook of Model Checking*, pp. 871–919 (2018)
11. Bruse, F., Falk, M., Lange, M.: The fixpoint-iteration algorithm for parity games. In: *Fifth International Symposium on Games, Automata, Logics and Formal Verification (GandALF)*. pp. 116–130 (2014)
12. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* **35**(8), 677–691 (1986)
13. Church, A.: Logic, arithmetic, and automata. In: *International Congress of Mathematicians (Stockholm, 1962)*, pp. 23–35. Institute Mittag-Leffler, Djursholm (1963)
14. Di Stasio, A., Murano, A., Vardi, M.Y.: Solving parity games: Explicit vs symbolic. In: *Implementation and Application of Automata: 23rd International Conference, CIAA 2018, Charlottetown, PE, Canada, July 30–August 2, 2018, Proceedings 23*. pp. 159–172. Springer (2018)
15. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 291–308. Springer International Publishing, Cham (2018)
16. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω -automata manipulation. In: *14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. pp. 122–129 (2016)
17. Ehlers, R.: Generalized Rabin(1) synthesis with applications to robust system synthesis. In: *Third International NASA Formal Methods Symposium (NFM)*. pp. 101–115 (2011)

18. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: 28th International Conference on Computer Aided Verification. pp. 333–339 (2016)
19. Ehlers, R., Schewe, S.: Natural colors of infinite words. In: 42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS) (2022), presentation available at <https://www.youtube.com/watch?v=RSd25TiELUo>
20. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: 21st International Conference on Computer Aided Verification (CAV). pp. 263–277 (2009)
21. Filiot, E., Jin, N., Raskin, J.F.: Antichains and compositional algorithms for ltl synthesis. *Formal Methods in System Design* **39**, 261–296 (2011)
22. Finkbeiner, B., Schewe, S.: Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.* **15**(5-6), 519–539 (2013)
23. Godhal, Y., Chatterjee, K., Henzinger, T.A.: Synthesis of AMBA AHB from formal specification: a case study. *Int. J. Softw. Tools Technol. Transf.* **15**(5-6), 585–601 (2013)
24. Gritzner, D., Greenyer, J.: Synthesizing executable PLC code for robots from scenario-based GR(1) specifications. In: *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops*. pp. 247–262 (2017)
25. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: *International Conference on Computer Aided Verification*. pp. 31–44. Springer (2006)
26. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! 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*. pp. 578–586. Springer (2018)
27. Piterman, N., Pnueli, A.: *Temporal Logic and Fair Discrete Systems*, pp. 27–73. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_2, https://doi.org/10.1007/978-3-319-10575-8_2
28. Piterman, N., Pnueli, A., Sa’ar, Y.: Synthesis of reactive (1) designs. In: *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8–10, 2006. Proceedings 7*. pp. 364–380. Springer (2006)
29. Pnueli, A.: The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science (FOCS)*. pp. 46–57 (1977)
30. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: *16th International Colloquium on Automata, Languages and Programming (ICALP)*. pp. 652–671 (1989)
31. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for the synthesis of reactive systems. *International Journal on Software Tools for Technology Transfer* **15**, 433–454 (2013)
32. Somenzi, F.: CUDD: CU Decision Diagram package release 3.0.0 (2016)
33. Walukiewicz, I.: Monadic second-order logic on tree-like structures. *Theoretical computer science* **275**(1-2), 311–346 (2002)
34. Wong, K.W., Kress-Gazit, H.: From high-level task specification to robot operating system (ROS) implementation. In: *First IEEE International Conference on Robotic Computing, IRC 2017, Taichung, Taiwan, April 10–12, 2017*. pp. 188–195 (2017)
35. Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., Murray, R.M.: Tulip: a software toolbox for receding horizon temporal logic planning. In: *14th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. pp. 313–314 (2011)

36. Zudaire, S.A., Nahabedian, L., Uchitel, S.: Assured mission adaptation of UAVs. *ACM Trans. Auton. Adapt. Syst.* **16**(3–4) (jul 2022)


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.





Knor: reactive synthesis using Oink

Tom van Dijk^(✉) , Feije van Abbema, and Naum Tomov

Formal Methods and Tools

University of Twente, Enschede, The Netherlands

`t.vandijk@utwente.nl`, `{f.vanabbema,n.tomov}@student.utwente.nl`

Abstract. We present an innovative approach to the reactive synthesis of parity automaton specifications, which plays a pivotal role in the synthesis of linear temporal logic. We find that our method efficiently solves the SYNTCOMP synthesis competition benchmarks for parity automata from LTL specifications, solving all 288 models in under a minute. We therefore direct our attention to optimizing the circuit size and propose several methods to reduce the size of the constructed circuits: (1) leveraging different parity game solvers, (2) applying bisimulation minimisation to the winning strategy, (3) using alternative encodings from the strategy to an and-inverter graph, (4) integrating post-processing with the ABC tool. We implement these methods in the Knor tool, which has secured us multiple victories in the PGAME track of the SYNTCOMP competition.

Keywords: Reactive synthesis · Parity games · Binary decision diagrams

1 Introduction

Reactive synthesis as first stated by Church [8,9] and outlined in [32] is the act of automatically constructing a reactive system such that all interactions with an unknown environment satisfy a linear temporal logic (LTL) specification. While early solutions were proposed to solve the synthesis problem via finite-state automata [7], until recently reactive synthesis using deterministic parity automata and parity games was deemed infeasible in practice, in part due to the lack of efficient translations from LTL to deterministic ω -automata. With the rise of direct translations, LTL synthesis tools such as *ltsynt* [27,33,34] and *Strix* [26] are capable of solving a wide range of specifications via deterministic parity automata and parity games, and perform better than some of the previous techniques avoiding deterministic parity automata.

The advantage of reactive synthesis is that synthesized systems are correct by construction and therefore do not need to be tested nor model checked for correctness. The reactive synthesis (SYNTCOMP) competition was founded to increase the impact of reactive synthesis in industry and improve the quality of synthesis tools [22,23]. Motivated by the new PGAME track in the SYNTCOMP competition, we seek to use the Oink parity game solver [11] in the competition and to implement the necessary infrastructure that translates the parity automata

of the competition into parity games suitable for Oink, and that translates the winning strategy computed by Oink into a Boolean circuit. We name this implementation **Knor**¹.

Knor leverages Oink to solve parity games with state-of-the-art parity game solvers [16], and the Sylvan binary decision diagrams (BDD) package [14] to implement most of the steps before and after solving and a purely symbolic parity game solver based on [25]. The techniques implemented in Knor have secured us multiple victories in the SYNTCOMP competition, in 2021, 2022 and 2023.

Following initial success of Knor in the competition, we observe a major difference with main competitors ltsynt and Strix. While Knor can solve all benchmarks in a remarkably short time, the constructed circuits are sometimes several orders of magnitude larger than the circuits constructed by other tools. Thus, we propose several techniques, mostly symbolic techniques that rely on binary decision diagrams, to reduce the size of the constructed circuits.

Contribution. We present the Knor tool that solves the synthesis problem of parity automata to Boolean circuits, built around the parity game solver Oink. We consider three methods to translate the given parity automaton to a parity game, and present a novel symbolic approach that improves upon an explicit translation by several orders of magnitude. As Oink implements several parity game solvers that have been shown in [16] to perform well for parity games derived from reactive synthesis benchmarks, we consider whether changing the algorithm impacts the size of the constructed circuit. We study whether applying bisimulation minimisation as in [15], which aims to minimize the number of states of the winning strategy after solving the parity game, can reduce the size of the circuits. Similarly, we study different encodings from the winning strategies into Boolean logic, in particular whether a onehot encoding of the states improves the circuit size. Finally, we apply a similar post-processing step as Strix by using the ABC tool [4,5] to minimize the constructed circuit after encoding it as an and-inverter graph. Sec. 3 describes Knor and provides accessible descriptions of the implemented techniques. We evaluate these techniques in Sec. 4. We discuss our findings in Sec. 5.

2 Preliminaries

Given two disjoint sets of Boolean variables I and O representing input and output signals, and an ω -regular language L of infinite words over the alphabet $2^{I \cup O}$ representing a specification, the reactive synthesis problem asks us to construct a controller that enforces L . The controller is a function $(2^{I \cup O})^* \times 2^I \rightarrow 2^O$ that yields a valuation of the output signals 2^O based on a history of input and output signals $(2^{I \cup O})^*$ and the current input signals 2^I .

While we are interested in the broader context of the synthesis of reactive systems that enforce specifications given in linear temporal logic (LTL), we

¹ Knor is the Dutch word for the sound that a pig makes, i.e., “oink”.

assume in this paper that L is given as a deterministic parity automaton. LTL specifications can be translated to a parity automaton of doubly-exponential size.

Deterministic parity automata (DPA) are ω -regular automata that accept ω -regular languages. A DPA is a tuple (Q, q_0, AP, Δ, F) , where Q is a finite set of states, $q_0 \in Q$ is the initial state, AP is a set of atomic propositions, $\Delta \subseteq Q \times 2^{AP} \times Q$ is the transition relation and $F: Q \rightarrow \mathbb{N}$ assigns to each state a *priority*. A *run* of the automaton is an infinite sequence of states consistent with the transition relation. A run is accepting if and only if the maximum priority that occurs infinitely often along the run is an even number. We define parity automata with priorities on states. Alternatively, priorities can also be on transitions.

A parity game is a DPA with two players Even and Odd, where the set of states Q is partitioned into two sets Q_0 and Q_1 . In this paper, we refer to the states of the parity game as vertices and the transitions of the parity game as edges. A run on a parity game is an infinite sequence of vertices where player Even decides the next vertex if the current vertex is in Q_0 , and player Odd if it is in Q_1 . A fundamental result for parity games is that they are memoryless determined [18], i.e., each vertex is winning for exactly one player, and both players have a positional strategy for each of their winning vertices.

To solve the synthesis problem, given a deterministic parity automaton over $AP = I \cup O$, we construct a parity game by *splitting* the automaton across I and O , letting one player (the environment) choose a valuation of variables in I and the other player (the controller) a valuation of variables in O .

The result of reactive synthesis is a Boolean circuit, structured as an and-inverter graph (AIG). An AIG is a directed acyclic graph, featuring terminal nodes that denote Boolean inputs (input signals and latches), internal nodes representing AND-gates, and edges with complementation for logical negation.

Binary decision diagrams [6, 17] (BDDs) are a well known data structure for representing and manipulating Boolean functions. A binary decision diagram is a rooted, directed acyclic graph. Its internal nodes represent decisions based on the values of Boolean variables, directing the path to one of the two child nodes, via the “true” edge (depicted as a solid arrow) and the “false” edge (depicted as a dashed arrow). Reaching the terminal node “1” indicates that the represented Boolean function evaluates to true for that particular valuation, and reaching the “0” node indicates a false evaluation. BDDs are recognized as a canonical representation of Boolean functions when they meet two conditions. First, they must be ordered; that is, they follow a fixed variable ordering when encountering Boolean variables. Second, they must be reduced, meaning that any redundant decision nodes with identical successors are eliminated [6]. BDDs can be incredibly efficient if a suitable variable ordering is found and the represented set is encoded in a way that results in small decision diagrams.

Multi-terminal binary decision diagrams (MTBDDs) extend BDDs by allowing terminal nodes to hold various types of data, not just the Boolean values true and false. The MTBDD implementation in Sylvan [14] in particular allows for terminal nodes to be labeled by 64-bit values. These labels can represent a wide

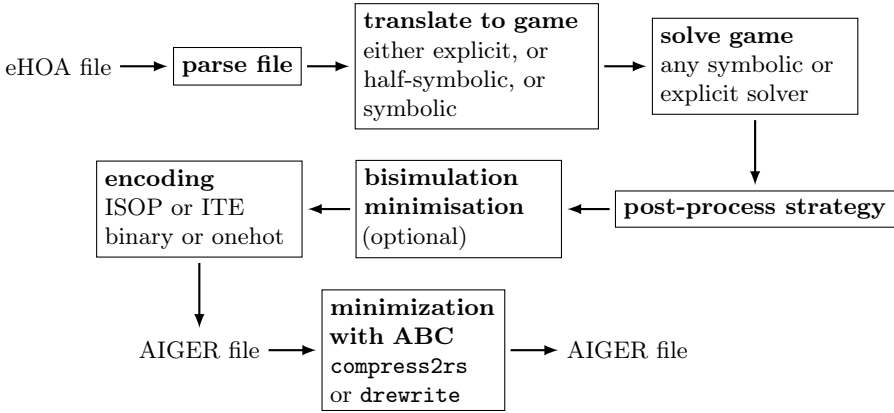


Fig. 1. Overview of Knor from input file to output file.

range of data, including 64-bit integers, pointers, floating-point numbers, or even pairs of 32-bit values.

3 Knor

We study reactive synthesis from parity automata to Boolean circuits in the **Knor** research tool. Knor is written in C++ and is publicly available under a permissive license via <https://www.github.com/trolando/knor>. See Fig. 1 for an overview of Knor. All steps of the program are discussed in the following sections.

3.1 Input format

Knor reads input files formatted using the extended Hanoi Omega-Automata (HOA) format [31].

The HOA format [1] is a file format to describe finite-state automata that accept sets of infinite words. The automata consist of a finite set of states Q , one or more initial states $I \subseteq Q$, a set of atomic propositions AP , and a labeled transition relation $\Delta \subseteq Q \times \mathbb{B}(AP) \times Q$, where each transition is labeled with a Boolean formula $\phi \in \mathbb{B}(AP)$, where we use $\mathbb{B}(AP)$ to denote the set of Boolean formulas over AP . Furthermore, the HOA format describes an acceptance condition of the automaton, i.e., a set of infinite runs of the automaton which are considered accepting. For the purposes of the current paper, we are only interested in the *parity condition*, i.e., the automaton is accepting if and only if the lowest/highest priority seen infinitely often along the run is even/odd, depending on whether the acceptance condition is *min even*, *min odd*, *max even* or *max odd*. In the HOA format, the priorities are either on states or on transitions.

The extended HOA format adds a distinction between controllable (output) and uncontrollable (input) atomic propositions [31].

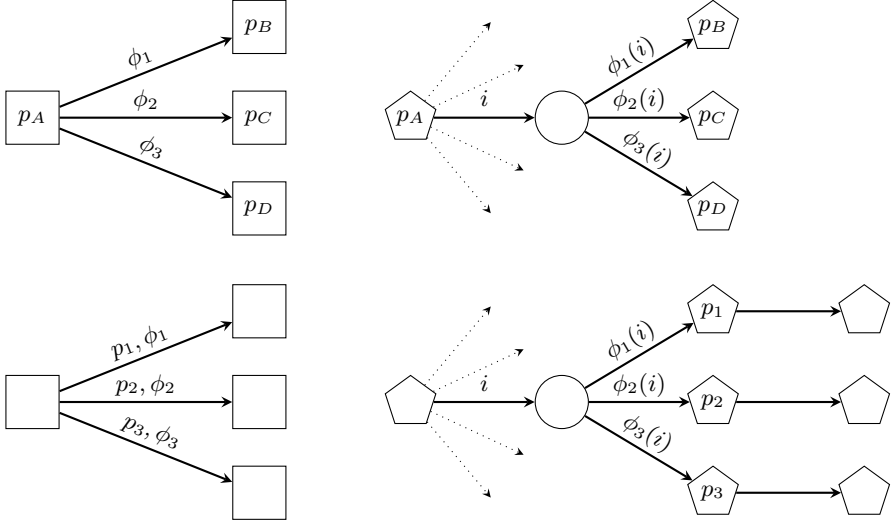


Fig. 2. Splitting a transition on the parity automaton (left) to construct the parity game (right), with priorities on the states (above) or on the transitions (below). We depict states by squares, vertices of the environment player by pentagons and vertices of the controller player by circles.

3.2 Output format

Knor can produce parity games in the standard PGSolver [20] format that is also accepted by Oink, as well as Boolean circuits in the AIGER format [3].

3.3 Translation from automaton to game

As described above, the parity automaton consists of a number of states with transitions labeled by a Boolean formula, and with the priorities either on the transitions or on the states.

To translate the automaton to a parity game, we need to split every transition into two parts. The environment player “moves first” by choosing a valuation of the input signals, and the controller player responds by setting output signals such that the specification is guaranteed. That is, the output signals are determined by the current state and the current input signals.

We propose three methods to convert the parity automaton to a parity game: a naive explicit method, a half-symbolic method and a fully symbolic method.

(Naive) Explicit method. The explicit method simply creates a parity game vertex for every state in the parity automaton, and then splits the transitions into two parts as in Fig. 2.

For every valuation i of the input signals, we create an intermediate vertex that is controlled by the controller player. This intermediate vertex should have the least relevant priority, typically 0. For every transition with a label (Boolean formula) that is satisfiable for i , we then create an edge from the intermediate vertex to the successor of the transition.

Since we want our parity games to have priorities on the vertices and not on the edges, we need to create extra vertices in case the automaton has priorities on transitions. This is also shown in Fig. 2. Priorities on the source vertex, intermediate vertex, and target vertices should be set to the least relevant priority (typically 0) or be ignored by the solver.

The result is an explicit parity game which Knor directly constructs using Oink. The game is then solved with any algorithm implemented by Oink.

Half-symbolic method. The fully explicit method works reasonably well for many of the smaller input models, however some models result in a significant exponential blowup of the parity game, as any game with n input signals has 2^n outgoing edges per source vertex. The extended HOA format actually encodes the labels on the transitions *symbolically* using Boolean formulas, so an exponential blowup in some cases can be expected. We propose a method that still results in an explicit game constructed using Oink, but that employs binary decision diagrams to reduce the number of intermediate vertices and extra transitions in the parity game.

For every state, we produce a multi-terminal binary decision diagram (MTBDD) encoding all outgoing transitions, with decision variables representing input signals ordered before variables representing output signals, and terminal nodes encoding both priority and successor state as a pair of two 32-bit numbers.

We then collect all **subroots** of the MTBDD after the input signals, i.e., along each path from the root node to a terminal node, we find the first node that is either a decision node with a variable of an output signal, or a terminal node. For every such node N , we create a corresponding intermediate vertex owned by the controller player. The paths leading to N correspond to valuations of the input signals that lead to that intermediate vertex, where the controller can decide how to respond. We let the controller choose to go to any state (vertex) encoded by a terminal node that is reachable from N . For every such terminal node, we simply add an edge from the intermediate vertex to the target vertex.

Fully-symbolic method. While the half-symbolic method already results in a major reduction in the size of the parity games, we can go further and encode the full transition relation of the parity automaton as a single BDD, which can then automatically be interpreted as a symbolic parity game simply by ordering variables as follows:

1. Variables s corresponding to the source state.
2. Variables i corresponding to input signals.
3. Variables o corresponding to output signals.

4. Variables p and s' corresponding to the priority (either from the transition or from the target state) and the target state.

One can read this BDD intuitively as follows: given some current state (1) and some current input values (2), if the controller sets certain output values (3) we arrive with some priority at our next state (4). Variables within these four groups can be ordered freely; however, we implement a naive approach and have not optimized this ordering; this is left as an opportunity for future work.

Since we encode the entire automaton as a single BDD, states that share some transitions can benefit from the automatic reduction offered by BDDs.

We present a translation from this symbolic parity game to an explicit parity game that explicitly uses the structure of the decision diagram to construct the game. This procedure consists of the following steps:

1. We create a **state vertex** controlled by the environment player for every state (with transitions) in the symbolic parity game. These vertices get priority 0.
2. Along each path in the BDD, we find the first decision node *after* the input signals. We create an **intermediate vertex** controlled by the controller player for every such node. These vertices also get priority 0.
3. Along each path in the BDD, we find the first decision node *after* the output signals. We decode the priority and the target state and create a **priority vertex** for the environment player with the decoded priority and with a single edge to the state vertex corresponding to the target state.
4. For every state, we compute the reachable decision nodes of step 2 and create edges from the state vertices to the intermediate vertices.
5. For every decision node of step 2, we compute the reachable decision nodes of step 3 and create edges from the intermediate vertices to the priority vertices.

Further improvements to this procedure are possible by considering that vertices may share many transitions, and additional vertices could be added based on the structure of the BDD. This could reduce the number of edges at the cost of more vertices. Furthermore, we do not merge the state vertices and priority vertices, which might reduce the number of vertices. This is left as an opportunity for future work.

3.4 Solving the parity game

Using the procedure described above, we can produce an explicit parity game that can be solved by Oink. As shown in [16], several solvers implemented in Oink are very efficient for parity games derived from reactive synthesis:

- strategy iteration (**psi**) [11,19]
- tangle learning (**tl**) [10]
- priority promotion (**npp**) [2,11]
- Zielonka’s recursive algorithm (**zlk**) [11,35]
- fixpoint iteration using freezing (**fpi**) [16]
- fixpoint iteration using justifications (**fpij**) [24]

We also implement a symbolic solver based on [25]. This symbolic solver implements fixpoint iteration with freezing using BDD operations, and operates directly on the BDD obtained by the fully-symbolic translation.

3.5 Post-processing the strategy

After applying the strategy to the symbolic parity game, we perform two post-processing steps. In the case that the strategy does not give all output signals a value, we default to setting output signals to false (or 0). We also compute all reachable vertices of the parity game from the initial state vertex, restricted to the winning strategy, and remove unreachable vertices.

3.6 Bisimulation minimisation

To further reduce the number of vertices of the parity game, we apply bisimulation minimisation. Bisimulation minimisation computes equivalence classes of vertices, i.e., all vertices that have the same behavior w.r.t. input and output signals. We use the signature-based partition refinement approach of [15].

Recall that the symbolic parity game is a BDD over the variables s, i, o, p, s' as described in Sec. 3.3. We first drop the priority variables p from the BDD, as the priorities on the states are not relevant after solving. We reserve fresh BDD variables c for the classes, which are ordered *after* the next state variables, i.e., $s < i < o < s' < c$. We maintain the current assignment from states to classes in a BDD over variables s' and c . The reason for s' rather than s is that this reduces the number of BDD operations. The initial partition assigns all states to a single equivalence class. We then repeatedly compute the current signature of all states, which is a BDD encoding for every state the *classes* that can be reached and the input/output values to reach them, as follows:

1. Given a BDD G encoding the symbolic parity game over the variables s, i, o, s' , and a BDD P encoding the current partition over the variables s' and c , we compute the BDD S representing the signatures over variables s, i, o, c by performing the operation `and_exists`(G, P, s').
2. We use the `refine` operation of [15] to replace the signatures (over variables i, o, c) in S by new classes, reusing previous class identifiers whenever possible, and renaming s variables to s' variables on-the-fly, resulting in the next BDD P over the variables s' and c .
3. We repeat steps 1 and 2 until the number of classes is stable.

Afterwards, we apply the obtained partition by replacing the states in the symbolic parity game by the equivalence classes.

3.7 Encoding the strategy as a circuit

There are several methods to create a Boolean circuit from the solver parity game. We first need to encode all reachable states of the parity game as latches in the

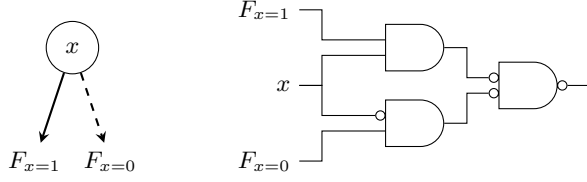


Fig. 3. Sketch of the encoding from a BDD decision node (left) to three AND-gates (right), representing the Boolean formula $(F_{x=1} \wedge x) \vee (\neg x \wedge F_{x=0})$.

Boolean circuit. We employ two methods for this: (1) one latch per state; and (2) one latch per BDD state variable. We call the former method **onehot** and the latter **binary**; in the first case at all times only a single latch is set, whereas in the second case the latches form a binary encoding of the states, similar to how they are encoded in the symbolic parity game. As the initial state of a Boolean circuit has all latches reset (to 0), we invert the latch that encodes the initial state for the **onehot** encoding and we encode the initial state as state 0 for the **binary** encoding.

We then compute a BDD F for every latch and for every output signal, where F is a BDD over the variables s, i (current state and current input signals) such that the latch or signal will be set if and only if F evaluates to true. We then translate each BDD F to an and-inverter graph. Again we propose two methods to achieve this:

- by using Shannon expansion (ITE) as in Fig. 3 recursively;
- by first obtaining the irredundant sum-of-products [28] (ISOP) of F in the form of a ZBDD [29], which can then directly be translated to an AIG: first all products are created, and then the products are connected through inverted AND-gates (as $ab \vee cd \equiv \neg(\neg(ab) \wedge \neg(cd))$).

We thus have four combinations: ITE with **binary** or **onehot** encoding and ISOP with **binary** or **onehot** encoding. Furthermore, we use a cache when creating AND-gates to avoid duplicate gates.

3.8 Post-processing with ABC

After encoding the strategy as a circuit, we apply optional post-processing of the circuit using ABC [5].

Similar to Strix, we apply the **compress2rs** script, which is described in [4]. The **compress2rs** script performs rewriting, refactoring, balancing, and truth-table-based resubstitution. While Strix applies the script until no further improvement is found, we halt when the improvement is less than 2.5%.

We also apply a sequence of three ABC commands, **drw**, **balance** and **drf**, which we call the **drewrite** script here. We apply this script until the improvement is less than 1%.

3.9 Usage of Knor

Knor expects an eHOA file on standard input; it also accepts a filename as a command line parameter instead. With the options `-a` and `-b`, Knor writes the constructed circuits to standard output as an AIGER file in ASCII or binary format respectively. With the option `-v`, Knor prints timings and other information to standard error.

By default, Knor uses the fully symbolic translation to a parity game. One can use `--naive` for the naive explicit encoding and `--explicit` for the half-symbolic encoding, and `--print-game` to print the resulting parity game in PGSolver format to standard output. Only the fully symbolic translation supports the full synthesis pipeline.

To choose an explicit-state solver of Oink, one can pick any solver from the list obtained with `--solvers`, in particular the solvers `--tl`, `--npp`, `--fpi`, `--fpj`, `--psi`, and `--zlk`. To solve using the symbolic solver, use `--sym`. With the option `--real`, Knor will only decide realizability and use tangle learning (`--tl`) as the default solver. The default solver for synthesis is the symbolic solver (`--sym`).

Bisimulation minimisation is applied by default, unless the `--no-bisim` option is used. To encode the circuit, Knor uses by default ITE and onehot encoding. To change this one can use the options `--isop` and `--binary`. To apply post-processing with ABC after constructing the circuits, use the options `--compress` and `--drewrite`.

4 Empirical Evaluation

We present the empirical results here.

4.1 Benchmarking

We evaluate the techniques implemented in Knor using the benchmarks of SYNTCOMP for the PGAME track that come from reactive synthesis, i.e., they are based on LTL specifications in the TLSF file format. In recent years, SYNTCOMP has also incorporated benchmarks in the PGAME track that do not come from reactive synthesis, such as artificial hard games that are designed to be time consuming for specific parity game solvers. Oink can easily handle such hard games by using a solver for which no hard game has been designed yet, and since our aim is to develop techniques for reactive synthesis specifically, we limit ourselves to benchmarks from the TLSF dataset². We also exclude input files that are not parity automata; this removes the `aut*.ehoa` files, two `test*.ehoa` files, and `UnderapproxStrengthenedDemo`, which is a Büchi automaton consisting of a single state. In total 288 input files remain.

The benchmarks are run on a machine with an Intel i5-13600KF processor. This is a 14-core processor, but we only use a single thread. Knor is compiled using gcc version 13.2.1. We repeat benchmarks 5 times and take the median to obtain

² https://github.com/SYNTCOMP/benchmarks/tree/v2023.4/parity/tlsf_based

Model	explicit	half-symbolic	symbolic
amba_decomposed_lock_15	T.O.	46	24
amba_decomposed_lock_14	T.O.	46	24
amba_decomposed_lock_13	T.O.	46	24
TwoCountersDisButA9	T.O.	668,065	7,249
amba_decomposed_lock_12	402,997,254	46	24
amba_decomposed_lock_11	100,820,998	46	24
amba_decomposed_lock_10	25,237,510	46	24
TwoCountersGui	21,022,475	256	155
TwoCountersDisButA8	15,254,863	497,310	4,721
full_arbiter_8	11,287,306	1,669,066	177,690
amba_decomposed_lock_9	6,323,718	46	24
amba_decomposed_encode_16	4,981,507	876	330
TwoCountersDisButA7	3,939,305	98,947	2,365
TwoCountersDisButA6	3,806,249	101,175	1,733

Table 1. Sizes in number of vertices of the largest parity games, sorted descending by size of parity games constructed using the explicit method.

Technique	Sum of Vertices	Time (sec)
explicit	622,987,565	1,177.91
half-symbolic	8,491,540	18.28
symbolic	620,510	11.76

Table 2. Cumulative size of parity games and time required for construction of the parity games of the 284 inputs that could be constructed by all three techniques.

the runtimes. All experimental scripts and log files are available as [12], and are also available online via <http://www.github.com/trolando/knor-experiments>.

4.2 Translating the parity automaton to a parity game

We first compare the three different techniques to obtain a parity game from the parity automaton: **explicit**, **half-symbolic** (only symbolic splitting) and **fully symbolic**.

Of the 288 benchmarks, the explicit method could not construct the parity game for four benchmarks within the timeout of 3600 seconds. See Table 1 for the largest parity games constructed by the explicit method, as well as the four input models for which no parity game could be constructed within 3600 seconds. The two other methods could construct the parity games within a reasonable amount of time, as is displayed in Table 2. The given time is only the time required for constructing the games and excludes time required for parsing the input file, which is the same for all methods.

Clearly, the fully symbolic method is superior to the other methods, both in the speed of construction and in the size of the constructed parity games. When

Solver	Circuit size		Time (sec)
	binary	onehot	
symbolic fpi (<code>--sym</code>)	317,403	122,514	18.45
fixpoint with justifications (<code>--fpj</code>)	350,035	139,900	0.16
fixpoint with freezing (<code>--fpi</code>)	353,120	140,297	0.22
strategy iteration (<code>--psi</code>)	334,149	140,916	0.57
priority promotion (<code>--npp</code>)	427,048	161,244	0.17
Zielonka (<code>--zlk</code>)	480,472	175,427	0.18
tangle learning (<code>--tl</code>)	604,044	213,632	0.17

Table 3. Cumulative circuit size in number of gates and cumulative solving time in number of seconds for the tested parity game solvers.

we consider individual input models, we find 20 cases where the half-symbolic approach results in slightly smaller parity games than the fully symbolic approach. The largest difference is 13 vertices (100 vertices instead of 113 vertices), which is negligible compared to the several orders of magnitude advantage that the fully symbolic method has in larger parity games, as Table 1 demonstrates. The cumulative time for the fully symbolic method is dominated by a handful of input models that require more than a second. Almost all parity games are constructed in fewer than 10 milliseconds.

Although the size of the parity game does not necessarily always correspond to the size of the constructed circuit or the required time for the entire synthesis process, it seems an obvious choice to only consider the fully symbolic translation in the remainder of this study.

4.3 Solving the parity game

We consider several parity game solvers, which have been shown in the past to be successful for solving games derived from synthesis: Zielonka’s recursive algorithm, priority promotion, tangle learning, the two fixpoint algorithms using freezing and justifications, strategy iteration, and symbolic fixpoint iteration. One of these, symbolic fixpoint iteration, directly operates on the symbolic parity game constructed by the fully symbolic method. All other solvers require the procedure outlined in Sec. 3.3 to translate the symbolic representation to an explicit game. The game is then solved, and we construct the circuit using the standard ITE encoding and either the binary or the onehot encoding of the states. We do not yet perform bisimulation minimisation or postprocessing using ABC.

The reason that it is interesting to consider different solvers is that different solvers may result in entirely different strategies to win the parity game. In particular, it may be that some solvers favor winning regions that reach either higher priorities or lower priorities, which can result in significant differences. This is in fact supported by the results presented here.

We report runtimes **for solving the parity games** (thus excluding time before solving and after solving) as well as the sizes of the circuits in Table 3.

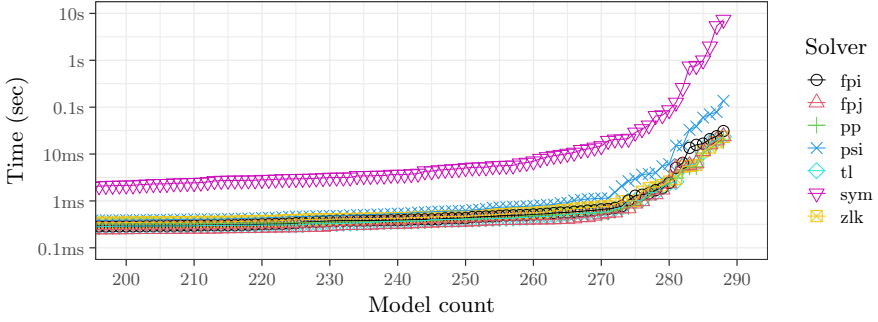


Fig. 4. Cactus plot of the number of parity games that can be solved within the given amount of time per solver.

Model	tl	sym	pp	psi	zlk	fpi	fpj
generalized_buffer_unreal1	0.02	7.36	0.02	0.14	0.02	0.03	0.02
generalized_buffer	0.01	5.37	0.01	0.07	0.01	0.02	0.01
genbuf2	0.01	1.98	0.01	0.03	0.01	0.01	0.01
full_arbiter_unreal3	0.00	1.00	0.00	0.06	0.00	0.02	0.01
amba_decomposed_arbiter_10	0.02	0.76	0.01	0.04	0.01	0.02	0.02
full_arbiter_8	0.02	0.74	0.02	0.08	0.02	0.02	0.02

Table 4. Overview of individual runtimes of each solver in seconds for the benchmarks for which at least one solver requires at least 500 milliseconds.

We observe that only the symbolic algorithm requires any time at all. The other algorithms each require less than a second to solve *all* benchmarks! When we consider the circuit sizes, the fully symbolic algorithm is superior with a cumulative 122,514 gates for all circuits. If we are interested in the best solver that solves all benchmarks in a fraction of a second, then clearly FPJ is the best algorithm, with a cumulative time of 0.16 seconds and a cumulative circuit size of 139,900 gates, although the difference with FPI is not that great.

Remarks. The solving time with the symbolic fixpoint iteration algorithm is dominated by just a few benchmarks. All algorithms solve the vast majority of parity games in a fraction of a second. See Fig. 4. Notice the logarithmic scale and that the vast majority of models are computed within a second for all solvers. Just a few models require more than 500 milliseconds to be solved, as is shown in Table 4.

We also did not take parallel operation into account. The symbolic FPI solver, the explicit FPI solver, and the strategy iteration solver have parallel implementations; the symbolic solver leverages the automatic parallelisation of decision diagram operations in Sylvan.

Solver	Circuit size		Time (sec)
	binary	onehot	
symbolic fpi (--sym) + minimisation	166,839	106,500	0.19
fixpoint with justifications (--fpj) + min.	205,937	124,489	0.15
symbolic fpi (--sym)	317,403	122,514	–
fixpoint with justifications (--fpj)	350,035	139,900	–

Table 5. Cumulative circuit size in number of gates and cumulative minimisation time in number of seconds for the symbolic fpi and the fixpoint with justifications solvers, with and without bisimulation minimisation after solving.

Solver	Encoding	Circuit size	Time
symbolic fpi (--sym)	ISOP, onehot	102,294	0.69
symbolic fpi (--sym)	ITE, onehot	106,500	0.61
fixpoint with justifications (--fpj)	ISOP, onehot	113,134	0.72
fixpoint with justifications (--fpj)	ITE, onehot	124,489	0.64
symbolic fpi (--sym)	ITE, binary	166,839	0.09
fixpoint with justifications (--fpj)	ITE, binary	205,937	0.12
symbolic fpi (--sym)	ISOP, binary	431,316	1.39
fixpoint with justifications (--fpj)	ISOP, binary	476,502	1.61

Table 6. Cumulative circuit size in number of gates and cumulative encoding time in seconds for the symbolic fpi and fixpoint with justification solvers, after bisimulation minimisation, using different encodings to obtain the circuit.

4.4 Bisimulation minimisation

We study the effects of bisimulation minimisation for the fully symbolic fixpoint iteration solver and for the explicit fixpoint iteration with justifications solver implemented in Oink.

As Table 5 shows, running bisimulation minimisation on the resulting strategy reduces the total circuit size in all cases. The required time to perform bisimulation minimisation is negligible with a cumulative time of a fraction of a second.

Bisimulation minimisation does not always improve the circuit size. There are a few cases where the procedure slightly increases the circuit size. There are also several models where the circuit size is reduced by several orders of magnitude. Interestingly, in some cases the circuit size is reduced to 0 AND-gates. It seems worthwhile to always apply bisimulation minimisation.

4.5 Encoding strategy to circuit

We now consider different encodings from the BDD of the strategy to the controller circuit. See Table 6. Surprisingly, the combination of ISOP and a binary encoding leads to a significantly worse result; whereas using ISOP with a onehot encoding slightly reduces the circuit sizes, but not by a significant amount.

Solver	Encoding	Method	Circuit size	Time
symbolic fpi (<code>--sym</code>)	ISOP	compress	61,434	149.26
symbolic fpi (<code>--sym</code>)	ITE	compress	62,506	121.27
fixpoint with justifications (<code>--fpj</code>)	ISOP	compress	71,240	125.29
fixpoint with justifications (<code>--fpj</code>)	ITE	compress	72,897	108.10
symbolic fpi (<code>--sym</code>)	ISOP	drewrite	80,077	58.72
symbolic fpi (<code>--sym</code>)	ITE	drewrite	80,425	53.21
fixpoint with justifications (<code>--fpj</code>)	ISOP	drewrite	80,454	60.88
fixpoint with justifications (<code>--fpj</code>)	ITE	drewrite	80,903	58.58
symbolic fpi (<code>--sym</code>)	ISOP		102,294	44.88
symbolic fpi (<code>--sym</code>)	ITE		106,500	39.81
fixpoint with justifications (<code>--fpj</code>)	ISOP		113,134	31.66
fixpoint with justifications (<code>--fpj</code>)	ITE		124,489	25.77

Table 7. Cumulative circuit size in number of gates for the two solvers, after bisimulation minimisation and using onehot encoding, then using different postprocessing methods to reduce circuit sizes. Given times are **total times** from parsing until writing, in seconds.

Tool	Circuit size	
	no post-processing	with post-processing
strix	68,550	41,314
sym-bisim-isop-onehot	87,823	50,624
ltsynt	544,804	98,996

Table 8. Cumulative size of the circuits for the 201 realizable inputs that could be constructed by all three tools, before and after post-processing with ABC.

Looking at individual benchmarks, we find that the most interesting differences occur with the `full_arbiter_*` and `amba_decomposed_arbiter_*` benchmarks. For these benchmarks, ISOP performs much worse than ITE with a binary encoding, but shows moderate improvement with the onehot encoding.

While there are some differences in the encoding times between the different approaches, the cumulative encoding time is less than two seconds in all cases.

4.6 Postprocessing with ABC

Finally, we apply postprocessing of the constructed circuit using ABC. See Table 7 for the results. We observe a very clear tradeoff of space and time. The best result is obtained by using the `compress` algorithm, which reduces the number of gates by about 40%, but this triples the runtime.

4.7 Comparison with other tools

We compare Knor to the tools Strix [26] and ltsynt [27,33,34]. We obtain the two competing tools from the SYNTCOMP 2023 artifact [21]. We use the following

command lines, similar to those used in the SYNTCOMP 2023 competition, to run the tools:

- Run Strix without post-processing in ABC:
`strix --auto --no-compress-circuit -t --hoa <filename>`
- Run Strix with post-processing in ABC:
`strix --auto -t --hoa <filename>`
- Run ltlsynt (without post-processing in ABC):
`ltlsynt --from-pgame=<filename> --aiger --verbose`

In the competition, ltlsynt had optional post-processing in ABC as part of the script rather than the executable. This script executed the following ABC commands: `collapse;strash;refactor;rewrite`. The Strix executable runs an embedded version of ABC, repeating the `compress2rs` script until no more improvement is found. To improve the fairness of the comparison, we change the post-processing for ltlsynt to start with `collapse;strash`, as this re-encoding of the circuit via binary decision diagrams significantly improves upon the circuit encoding by ltlsynt, followed by repeating the `compress2rs` script until there is no more improvement. This gives better results than obtained by ltlsynt in SYNTCOMP 2023.

Only 208 of the 288 input files are realizable. Of these, Strix did not solve the following inputs within the 3600 seconds time limit: `amba_decomposed_lock_14`, `amba_decomposed_lock_15`, `Automata325`, `GameLogic`, `genbuf2`, `SPIPureNext`, `generalized_buffer`. Except for `amba_decomposed_lock_15`, ltlsynt solved all inputs. Disregarding inputs that could not be solved by Strix or ltlsynt, we have 201 realizable inputs that can be solved within the time limit by all three tools. We provide the results with and without post-processing using ABC in Table 8. Considering individual results, we observe that Strix yielded smaller circuits in 142 cases (147 with post-processing) and Knor yielded smaller circuits in 47 cases (also 47 with post-processing). For the larger circuits, the `amba_decomposed_arbiter_*` inputs favored Knor (1527 vs 8282 gates, after post-processing), while Strix did better on the `full_arbiter_` inputs (1594 vs 26040 gates, after post-processing).

Table 8 clearly shows that all tools benefit from the post-processing. While Strix gives the best results for circuit size, the cumulative circuit size of Knor is only 23% more. Knor solves the entire set of inputs, including post-processing by ABC, in about 2.5 minutes, while Strix and ltlsynt cannot solve some benchmarks within the time limit of 1 hour, before post-processing.

5 Discussion

In this work, we studied techniques to improve reactive synthesis of parity automata to Boolean circuits using a new tool named **Knor**. We proposed a number of techniques and empirically evaluated these techniques using the benchmarks of the SYNTCOMP competition derived from LTL specifications. Knor has won the PGAME track of the competition several times.

The evidence presented in the empirical evaluation suggests that the best approach for deciding **realizability** is to use the fully symbolic translation from parity automaton to parity game, and any fast explicit-state parity game solver (like a tangle learning variation) for which no hard games have yet been designed. The latter is only needed to counteract any efforts aimed at impairing Knor’s performance in SYNTCOMP through the introduction of artificially difficult benchmarks.

For **synthesis**, considering a low circuit size as our primary objective, the clear solution is to use either symbolic fpi (`--sym`) or fixpoint with justifications (`--fpj`), preferring the former at the cost of speed in a few benchmarks, always apply bisimulation minimisation (`--bisim`), use a onehot encoding (`--onehot`) with either ITE or ISOP encoding, and apply postprocessing using ABC’s `compress2rs` script (`--compress`).

Knor is publicly available via <https://www.github.com/trolando/knor>.

Future work There are many opportunities for future improvements to the entire pipeline. We already mentioned playing with the variable ordering within the variable groups of the symbolic parity game, and considering slightly more efficient translations from the symbolic parity game to an explicit game in Oink.

We could also consider designing a parity game solving algorithm that explicitly results in small strategies. Some solvers might yield a multi-strategy, where multiple edges in the parity game can be taken to win the game. This could potentially be exploited to simplify the circuits.

It may also be useful to consider bisimulation minimisation on the parity game before solving, and to change the encoding of the states into the BDD, as we currently use a naive binary encoding of the state identifiers in the eHOA format. There may also be other encoding strategies to obtain the Boolean circuit, such as a different encoding of the latches or the approach of [30].

Beyond the reactive synthesis of parity automaton specifications, we may also explore symbolic techniques, including those outlined in this paper, for the synthesis of LTL specifications, building on the preliminary results from our earlier prototype described in [13].

Acknowledgements

We thank Alan Mishchenko for his helpful comments on using ABC for Boolean circuit minimisation. The first author is supported by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 893732.

Data availability statement

The software, benchmarks and analysed dataset are available as [12]. In addition, the version of Knor studied in the current paper is tagged in the Github repository of Knor as: <https://github.com/trolando/knor/tree/TACAS24>.

References

1. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi Omega-Automata Format. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (2015)
2. Benerecetti, M., Dell’Erba, D., Mogavero, F.: Solving Parity Games via Priority Promotion. In: CAV 2016. LNCS, vol. 9780, pp. 270–290. Springer (2016)
3. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Formal Models and Verification, Johannes Kepler University (2011), <https://fmv.jku.at/papers/BiereHeljankoWieringa-FMV-TR-11-2.pdf>
4. Brayton, R., Mishchenko, A.: Scalable logic synthesis using a simple circuit structure. In: Proc. of Internal Workshop on Logic Synthesis. vol. 6, pp. 15–22 (2006)
5. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 24–40. Springer (2010)
6. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
7. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society **138**, 295–311 (1969)
8. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Summaries of the Summer Institute of Symbolic Logic **1**, 3–50 (1957)
9. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians. pp. 23–35 (1962)
10. van Dijk, T.: Attracting tangles to solve parity games. In: CAV (2). Lecture Notes in Computer Science, vol. 10982, pp. 198–215. Springer (2018)
11. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 291–308. Springer (2018)
12. van Dijk, T.: Artifact of Knor: reactive synthesis using Oink (2023). <https://doi.org/10.4121/8794d8c0-5959-42f9-ba34-68f2137145a7>
13. van Dijk, T., Abraham, R., Sickert, S.: Almost-symbolic synthesis via delta-2-normalisation for linear temporal logic. In: 10th Workshop on Synthesis (2021)
14. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. J. Softw. Tools Technol. Transf. **19**(6), 675–696 (2017)
15. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. Int. J. Softw. Tools Technol. Transf. **20**(2), 157–177 (2018)
16. van Dijk, T., Rubbens, B.: Simple fixpoint iteration to solve parity games. In: GandALF. EPTCS, vol. 305, pp. 123–139 (2019)
17. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Int. J. Softw. Tools Technol. Transf. **3**(2), 112–136 (2001)
18. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS. pp. 368–377. IEEE Computer Society (1991)
19. Fearnley, J.: Efficient parallel strategy improvement for parity games. In: CAV (2). LNCS, vol. 10427, pp. 137–154. Springer (2017)
20. Friedmann, O., Lange, M.: Solving parity games in practice. In: ATVA. LNCS, vol. 5799, pp. 182–196. Springer (2009)
21. Jacobs, S., Perez, G., Schlehuber-Caissier, P.: Data, scripts, and results from SYNTCOMP 2023. Zenodo (2023). <https://doi.org/10.5281/zenodo.8161423>
22. Jacobs, S., Bloem, R.: The reactive synthesis competition: SYNTCOMP 2016 and beyond. In: SYNT@CAV. EPTCS, vol. 229, pp. 133–148 (2016)

23. Jacobs, S., Pérez, G.A., Abraham, R., Bruyère, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Lüttenberger, M., Meyer, K.J., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (SYNTCOMP): 2018-2021. *CoRR* **abs/2206.00251** (2022)
24. Lapauw, R., Bruynooghe, M., Denecker, M.: Improving parity game solvers with justifications. In: *VMCAI. Lecture Notes in Computer Science*, vol. 11990, pp. 449–470. Springer (2020)
25. Lijzenga, O., van Dijk, T.: Symbolic parity game solvers that yield winning strategies. In: *GandALF. EPTCS*, vol. 326, pp. 18–32 (2020)
26. Lüttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica* **57**(1-2), 3–36 (2020)
27. Michaud, T., Colange, M.: Reactive synthesis from ltl specification with spot. In: *Proceedings of the 7th Workshop on Synthesis, SYNT@CAV*. vol. 5 (2018)
28. Minato, S.: Fast generation of prime-irredundant covers from binary decision diagrams. *IEICE transactions on fundamentals of electronics, communications and computer sciences* **76**(6), 967–973 (1993)
29. Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: *DAC*. pp. 272–277. ACM Press (1993)
30. Miyasaka, Y., Mishchenko, A., Wawrzynek, J., Fraser, N.J.: Synthesizing a class of practical boolean functions using truth tables. In: *31st International Workshop on Logic and Synthesis* (2022)
31. Pérez, G.A.: The extended HOA format for synthesis. *CoRR* **abs/1912.05793** (2019)
32. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: *POPL*. pp. 179–190. ACM Press (1989)
33. Renkin, F., Schlehuber, P., Duret-Lutz, A., Pommellet, A.: Improvements to ltlsynt. In: *10th Workshop on Synthesis* (2021)
34. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Dissecting ltlsynt. *Formal Methods in System Design* (2023). <https://doi.org/10.1007/s10703-022-00407-6>
35. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.* **200**(1-2), 135–183 (1998)

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.





On Dependent Variables in Reactive Synthesis

S. Akshay¹(✉), Ilyahu Basa², Supratik Chakraborty¹, and Dror Fried²

¹ IIT Bombay, Mumbai, India

akshayss@cse.iitb.ac.in

² The Open University of Israel, Ra'anana, Israel

Abstract. Given a Linear Temporal Logic (LTL) formula over input and output variables, reactive synthesis requires us to design a deterministic Mealy machine that gives the values of outputs at every time step for every sequence of inputs, such that the LTL formula is satisfied. In this paper, we investigate the notion of dependent variables in the context of reactive synthesis. Inspired by successful pre-processing techniques in Boolean functional synthesis, we define dependent variables in reactive synthesis as output variables that are uniquely assigned, given an assignment to all other variables and the history so far. We describe an automata-based approach for finding a set of dependent variables. Using this, we show that dependent variables are surprisingly common in reactive synthesis benchmarks. Next, we develop a novel synthesis framework that exploits dependent variables to construct an overall synthesis solution. By implementing this framework using the widely used library **Spot**, we show that reactive synthesis that exploits dependent variables can solve some problems beyond the reach of existing techniques. Furthermore, we observe that among benchmarks with dependent variables, if the count of non-dependent variables is low (≤ 3 in our experiments), our method outperforms state-of-the-art tools for synthesis.

Keywords: Reactive synthesis · Functionally dependent variables · BDDs

1 Introduction

Reactive synthesis concerns the design of deterministic transducers (often Mealy or Moore machines) that generate a sequence of outputs in response to a sequence of inputs such that a given temporal logic specification is satisfied. Church introduced the problem [12] in 1962, and there has been a rich and storied history of work in this area over the past six decades. Recently, it was shown that a form of pre-processing, viz. decomposing a Linear Temporal Logic (LTL) specification, can lead to significant performance gains in downstream synthesis steps [15]. The general idea of pre-processing a specification to simplify synthesis has also been used very effectively in the context of Boolean functional synthesis [4,5,17,18,25]. Motivated by the success of one such pre-processing step, viz. identification of uniquely defined outputs, in Boolean functional synthesis, we introduce the notion of dependent outputs in the context of reactive synthesis in this paper. We develop its theory and show by means of extensive experiments that dependent outputs are common in reactive synthesis benchmarks, and can be effectively

exploited to obtain synthesis techniques with orthogonal strengths vis-a-vis existing state-of-the-art techniques.

In the context of propositional specifications, it is not uncommon for a specification to uniquely define an output variable in terms of the input variables and other output variables. A common example of this arises when auxiliary variables, called Tseitin variables, are introduced to efficiently convert a specification not in conjunctive normal form (CNF) to one that is in CNF [28]. Being able to identify such uniquely defined variables efficiently can be very helpful, whether it be for checking satisfiability, for model counting or synthesis. This is because these variables do not alter the basic structure or cardinality of the solution space of a specification regardless of whether they are projected out or not. Hence, one can often simplify the reasoning about the specification by ignoring (or projecting out) these variables. In fact, the remarkable practical success of Boolean functional synthesis tools such as Manthan [18] and BFSS [4, 5] can be partly attributed to efficient techniques for identifying a large number of uniquely defined variables. We draw inspiration from these works and embark on an investigation into the role of uniquely defined variables, or *dependent variables*, in the context of reactive synthesis. To the best of our knowledge, this is the first attempt at directly using dependent variables for reactive synthesis.

We start by first defining the notion of dependent variables in LTL specifications for reactive synthesis. Specifically, given an LTL formula φ over a set of input variables I and output variables O , a set of variables $X \subseteq O$ is said to be *dependent* on a set of variables $Y \subseteq I \cup (O \setminus X)$ in φ , if at every step of every infinite sequence of inputs and outputs satisfying φ , the finite history of the sequence together with the current assignment for Y uniquely defines the current assignment for X . The above notion of dependency generalizes the notion of uniquely defined variables in Boolean functional synthesis, where the value of a uniquely defined output at any time is completely determined by the values of inputs and (possibly other) outputs at that time. We show that our generalization of dependency in the context of reactive synthesis is useful enough to yield a synthesis procedure with improved performance vis-a-vis competition-winning tools, for a non-trivial number of reactive synthesis benchmarks.

We present a novel automata-based technique for identifying a subset-maximal set of dependent variables in an LTL specification φ . Specifically, we convert φ to a language-equivalent non-deterministic Büchi automaton (NBA) A_φ , and then deploy practically efficient techniques to identify a subset-maximal set of outputs X that are dependent on $Y = I \cup (O \setminus X)$. We implemented our method to determine the prevalence of dependent variables in existing reactive synthesis benchmarks. Our finding shows that out of 1141 benchmarks taken from the SYNTCOMP [21] competition, 300 had at least one dependent output variable and 26 had all output variables dependent.

Once a subset-maximal set, say X , of dependent variables is identified, we proceed with the synthesis process as follows. Referring to the NBA A_φ alluded to above, we first transform it to an NBA A'_φ that accepts the language L' obtained from $L(\varphi)$ after removing (or projecting out) the X variables. Our

experiments show that A'_φ is more compactly representable compared to A_φ , when using BDD-based representations of transitions (as is done in state-of-the-art tools like **Spot** [7]). Viewing A'_φ as a new (automata-based) specification with output variables $O \setminus X$, we now synthesize a transducer T_Y from A' using standard reactive synthesis techniques. This gives us a strategy $f^Y : \Sigma_I^* \rightarrow \Sigma_{O \setminus X}$ for each non-dependent variable in $O \setminus X$. Next, we use a novel technique based on Boolean functional synthesis to directly construct a circuit that implements a transducer T_X that gives a strategy $f_X : \Sigma_Y^* \rightarrow \Sigma_X$ for the dependent variables. Significantly, this circuit can be constructed in time polynomial in the size of the (BDD-based) representation of A_φ . The transducers T_Y and T_X are finally merged to yield an overall transducer T that describes a strategy $f : \Sigma_I^* \rightarrow \Sigma_O$ solving the synthesis problem for φ .

We implemented our approach in a tool called **DepSynt**. Our tool is developed in C++ using APIs from the widely used library **Spot** for representing and manipulating non-deterministic Büchi automata. We performed a comparative analysis of our tool with winning entries of the SYNTCOMP [21] competition to evaluate how knowledge of dependent variables helps reactive synthesis. Our experimental results show that identifying and utilizing dependent variables results in improved synthesis performance when the count of non-dependent variables is low. Specifically, our tool outperforms state-of-the-art and highly optimized synthesis tools on benchmarks that have at least one dependent variable and at most 3 non-dependent variables. This leads us to hypothesize that exploiting dependent variables benefits synthesis when the count of non-dependent variables is below a threshold. Given the preliminary and un-optimized nature of our implementation, we believe there is significant scope for improvement.

Related work. Reactive synthesis has been an extremely active research area for the last several decades (see e.g. [9, 12, 15, 16, 24]). Not only is the theoretical investigation of the problem rich, there are also several tools that are available to solve synthesis problems in practice. These include solutions like **l1tsynt** [23] based on **Spot** [7], **Strix** [22] and **BoSY** [14]. Our tool relies heavily on **Spot** and its APIs, which we use liberally to manipulate non-deterministic Büchi automata. Our synthesis approach is based on the standard conversion of LTL formula to NBA, and then from NBA to deterministic parity automata (DPA) (see [8] for an overview of the challenges of reactive synthesis).

Our work may be viewed as lifting the idea of uniquely defined variables used in Boolean functional synthesis to the context of reactive synthesis. Viewed from this perspective, our work is not the first to lift ideas from Boolean functional synthesis to the reactive context. Following an approach for Boolean functional synthesis that decomposes a specification into separate formulas on input variables and on output variables [11], the work in [6] constructed a reactive synthesis tool for specific benchmarks that admit a separation of the specification into formulas for only environment variables and formulas for only system variables. The current work serves as an additional example in support of the hypothesis that intuition from Boolean functional synthesis can be helpful and effective in the reactive synthesis context.

The remainder of the paper is structured as follows. We introduce definitions and notations in Section 2. In Section 3 we define dependent variables for LTL formulas, and describe an algorithm to find them. In Section 4 we describe our automata-based synthesis framework and discuss its implementation details in Section 5. We describe our evaluation in Section 6 and conclude in Section 7. Missing proofs and additional experiments can be found in the full-version [2].

2 Preliminaries

Given a finite alphabet Σ , an infinite *word* w is a sequence $w_0w_1w_2\cdots$ where for every i , the i^{th} letter of w , denoted w_i , is in Σ . The *prefix* $w_0\cdots w_i$ (of size $i+1$) of w is denoted by $w[0, i]$. Note that $w[0, 0] = w_0$. We use $w[0, -1]$ to denote the empty word. The set of all infinite words over Σ is denoted by Σ^ω . We call $L \subseteq \Sigma^\omega$ a *language* over infinite words in ω . For our work, the alphabet Σ is often the product of two distinct alphabets Σ_X and Σ_Y , i.e. $\Sigma = \Sigma_X \times \Sigma_Y$. In such cases, for every $a = (a_1, a_2) \in \Sigma$, we abuse notation and use $a.X$ to denote the projection of a on Σ_X , i.e. the letter $a_1 \in \Sigma_X$. Similarly, $a.Y$ denotes the projection of a on Σ_Y , i.e. the letter $a_2 \in \Sigma_Y$. For an infinite word $w \in \Sigma^\omega$, we use $w.X$ to denote the infinite word in Σ_X^ω obtained by projecting each letter in w on Σ_X i.e. $w.X = w_0.Xw_1.X\cdots$.

Linear Temporal Logic. A Linear Temporal Logic (LTL) formula is constructed with a finite set of propositional variables V , using Boolean operators such as \vee, \wedge , and \neg , and temporal operators such as next (X), until (U), etc. The set V induces an alphabet $\Sigma_V = 2^V$ of all possible assignments (*true/false*) to the variables of V . The semantics of the operators and satisfiability relation are defined as usual [20]. The language of an LTL formula φ , denoted $L(\varphi)$ is the set of all words in Σ_V^ω that satisfy φ . For an LTL formula φ over V , we use $|V|$ to denote the number of variables in V , and $|\varphi|$ to denote the size of the formula, i.e., count of its subformulas. For clarity of exposition, we sometimes abuse notation and identify the singleton variable set $\{z\}$ with z . We also use Σ for Σ_V , when V is clear from the context.

Nondeterministic Büchi Automata. A Nondeterministic Büchi Automaton (NBA) is a tuple $A = (\Sigma, Q, \delta, q_0, F)$ where Σ is the alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a non-deterministic transition function, q_0 is the initial state and $F \subseteq Q$ is a set of accepting states. Automaton A can be seen as a directed labeled graph with vertices Q and an edge (q, q') exists with a label a if $q' \in \delta(q, a)$. We denote the set of incoming edges to q by $in(q)$ and the set of outgoing edges from q by $out(q)$. A *path* in A is a (possibly infinite) sequence of states $\rho = (q_{i_0}, q_{i_1}, \cdots)$ in which for every $j > 0$, $(q_{i_j}, q_{i_{j+1}})$ is an edge in A . A *run* is a path that starts in q_0 , and is *accepting* if it visits a state in F infinitely often. A *word* $w = \sigma_{i_0}\sigma_{i_1}\cdots$ induces a run $\rho = (q_{i_0}, q_{i_1}, \cdots)$ of A if $q_{i_0} = q_0$ and for every $j \geq 0$, $q_{i_{j+1}} \in \delta(q_{i_j}, \sigma_{i_j})$. Since A is nondeterministic, a word can have many runs. A word is *accepting* if it has an accepting run in A . The language

$L(A)$ is the set of all accepting words in A . Wlog, we assume that all states and edges that are not a part of any accepting run (i.e. do not reach a cycle with an accepting state) are removed. This can be done by a simple pre-processing pass on the NBA. Finally, every LTL formula φ can be transformed in time exponential time in the size of φ to an NBA A_φ for which $L(\varphi) = L(A_\varphi)$ [20, 29]. When φ is clear from the context we omit the subscript and refer to A_φ as A . We denote by $|A|$ the size of an automaton, i.e., number of its states and transitions.

Reactive Synthesis. A *reactive LTL formula* is an LTL formula φ over a set of input variables I and output variables O , with $I \cap O = \emptyset$. In *reactive synthesis* we are given a reactive LTL formula φ , and the challenge is to synthesize a function, called *strategy*, $f : \Sigma_I^* \rightarrow \Sigma_O$ such that every word $w \in (\Sigma_I \times \Sigma_O)^\omega$ obtained by using this strategy at every time step is in $L(\varphi)$. If such a strategy exists we say that φ is *realizable*. Otherwise, we say that φ is *unrealizable*. In what follows, we always consider only reactive LTL formulas and hence omit the "reactive" prefix while referring to them. The synthesized strategy $f : \Sigma_I^* \rightarrow \Sigma_O$ is typically described (explicitly or symbolically) as a *transducer* $T = (\Sigma_I, \Sigma_O, S, s_0, \delta, \lambda)$ in which Σ_I and Σ_O are input and output alphabet respectively, S is a set of states with an initial state s_0 , $\delta : S \times \Sigma_I \rightarrow S$ is a deterministic transition function, and $\lambda : S \times \Sigma_I \rightarrow \Sigma_O$ is the output function. A standard procedure in solving reactive synthesis is to transform a given LTL formula φ to an NBA A_φ for which $L(A_\varphi) = L(\varphi)$. Subsequently, A_φ is transformed to a Deterministic Parity Automata (DPA) that turns to a parity game, whose solution is described as a transducer T_{A_φ} . As the following theorem shows, this approach incurs a double exponential blowup in the worst-case.

Theorem 1. 1. *Reactive synthesis can be solved in $O(2^{n \cdot 2^n})$, where n is the size of the LTL formula.*
 2. *Given an NBA A with n states, computing transducer T_A takes $\Omega(2^{n \log n})$.*

3 Dependent variables in reactive LTL

We begin by defining dependent variables for (reactive) LTL formulas and propose an algorithm for finding a maximal set of dependent variables. While there are several notions of dependency that can be considered, we discuss one that we have found to be useful in reactive synthesis. Specifically, we require that the value of a dependent output variable be completely determined by the values of inputs and other output variables and their finite history at every step of the interaction between the reactive system and its environment. We consider dependencies restricted to output variables, since having dependent input variables would preclude some input sequences, rendering the specification unrealizable.

Definition 1 (Variable Dependency in LTL). *Let φ be an LTL formula over V with input variables $I \subseteq V$ and output variables $O = V \setminus I$. Let X, Y be disjoint sets of variables where $X \subseteq O$. We say that X is dependent on Y in φ if for every pair of words $w, w' \in L(\varphi)$ and $i \geq 0$ if $w[0, i-1] = w'[0, i-1]$ and*

$w_i.Y = w'_i.Y$, then we have $w_i.X = w'_i.X$. Further, we say that X is dependent in φ if X is dependent on $V \setminus X$ in φ , i.e., it is dependent on all the remaining variables.

Note that two words in $L(\varphi)$ with different prefixes can have different values for X for the same values for Y , if X is dependent on Y . Also, observe that if X is dependent on Y in φ for some Y , then it is also dependent in φ .

As an example, consider an LTL formula φ with input variable y and output variable x . The corresponding input and output alphabets are $\Sigma_X = \{x, \neg x\}$ and $\Sigma_Y = \{y, \neg y\}$ respectively. Suppose $L(\varphi) = \{w^1, w^2, w^3\}$ where $w^1 = (y, x)^\omega$, $w^2 = (\neg y, x)^\omega$ and $w^3 = (y, x)(\neg y, x)(y, \neg x)^\omega$. Then x is dependent on y in φ . Specifically, note that $w^1[0, 1] \neq w^3[0, 1]$, and hence the dependency of x is not violated although $w^1_2.y = w^3_2.y$ and $w^1_2.x \neq w^3_2.x$.

3.1 Maximally dependent sets of variables Given an LTL formula $\varphi(I, O)$, we say that a set $X \subseteq O$ is a *maximal dependent set* in φ if X is dependent in φ and every set of outputs that strictly contains X is not dependent in φ . As in the propositional case [27], finding maximum or minimum dependent sets is intractable, hence we focus on subset-maximality. Given a variable z and set Y , checking whether z is dependent on Y , can easily be used to finding a maximal dependent set. Indeed, we would just need to start from the empty set and iterate over output variables, checking for each if it is dependent on the remaining variables. We give the pseudocode for this in [2]. Note that when all output variables are not dependent, the order in which output variables are chosen may play a significant role in the size of the maximal set obtained. We currently use a naive ordering (first appearance), and leave the problem of better heuristics for getting larger maximal independent sets to future work.

3.2 Finding dependent variables via automata As explained above, the heart of the dependency check is to verify whether a given output variable is dependent on a set of other variables. We now develop an approach for doing so based on the nondeterministic Büchi automaton A_φ that represents the same language as the LTL formula φ . Our framework uses the notion of compatible pairs of states of the automaton:

Definition 2. Let $A = (\Sigma, Q, \delta, q_0, F)$ be an NBA with states s, s' in Q . Then the pair (s, s') is *compatible* in A if there are runs from q_0 to s and from q_0 to s' on the same word $w \in \Sigma^*$.

Recall that in our definition, only states and edges that are part of an accepting run exist in A . Then we have the following definition.

Definition 3. Let φ be an LTL formula over V with input variables $I \subseteq V$ and output variables $O = V \setminus I$. Let X, Y be disjoint sets of variables where $X \subseteq O$. Let A_φ be an NBA that describes φ . We say that X is *automata dependent* on Y in A_φ , if for every pair of compatible states s, s' and assignments σ, σ' for V , where $\sigma.Y = \sigma'.Y$ and $\sigma.X \neq \sigma'.X$, $\delta(s, \sigma)$ and $\delta(s, \sigma')$ cannot both exist in A_φ . We say that X is *automata dependent* in A_φ if X is automata dependent on Y in A_φ and $Y = V \setminus X$.

As an example, consider NBA A_1 in Figure 1, constructed from some LTL formula with input $I = \{i\}$ and outputs $O = \{o_1, o_2\}$. For notational simplicity, we use $\Sigma_I = \{0, 1\}$, $\Sigma_O = \{0, 1\}^2$, and edges are labeled by values of $(i, o_1 o_2)$. It is easy to see that $(q_0, q_0), (q_1, q_1)$ are compatible pairs, but so are $(q_0, q_1), (q_1, q_0)$ since both q_0 and q_1 be reached from the initial state on reading the word $(0, 00)(0, 00)$ of length 2. Now consider output o_1 . It is not dependent on $\{i\}$, i.e., only the input, since from q_0 with $i = 0$, we can go to different states with different values of o_1 . But o_1 is indeed dependent on $\{i, o_2\}$. To see this consider every pair of compatible states – in this case all pairs. Then if we fix the values of i and o_2 , there is a unique value of o_1 that permits state transitions to happen from the compatible pair. For example, regardless of which state we are in, if $i = 0, o_2 = 0$, o_1 must be 0 for a state transition to happen. On the other hand, o_2 is not dependent on either $\{i\}$ or $\{i, o_1\}$ (as can be seen from (q_0, q_1) with $i = 1, o_1 = 1$). The following theorem relates automata-based dependency and dependency in LTL (for proof, see [2]), allowing us to focus only on the former.

Theorem 2. *Let φ be an LTL formula with set of variables $V = I \cup O$, where $X \subseteq O$ and $Y \subseteq I \cup (O \setminus X)$. Let A_φ be an NBA with $L(\varphi) = L(A_\varphi)$. Then X is dependent on Y in φ if and only if X is automata dependent on Y in A_φ .*

Finding Compatible States. We find all compatible states in an automaton in Algorithm 1 as follows. We maintain a list of in-process compatible pairs C that is initialized with (q_0, q_0) – an undoubtedly compatible pair. At each step, until C becomes empty, we pick a pair $(s_i, s_j) \in C$, add it to the compatible pair set P , and remove it from C (in line 4). Then (in lines 5-8), we check (in line 6) if outgoing transitions from (s_i, s_j) lead to a new pair (s'_i, s'_j) not already in P or C , that can be reached on reading the same letter σ . If so, we add this pair to the in-process set C . All pairs that we add to P, C are indeed compatible, and nothing is removed from P . When the algorithm terminates, C is empty, which means all possible ways (from initial state pair) to reach a compatible pair have been explored, thus showing correctness.

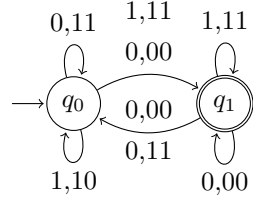


Fig. 1. An Example NBA A_1

Finally, we show how to check dependency using automata, by implementing procedure **isAutomataDependent**, shown in Algorithm 2. This procedure takes an NBA A_φ , a candidate dependent output z and a candidate dependency set $Y \subseteq V \setminus \{z\}$ as inputs, and tries to find a witness to z *not* being dependent on Y . If no such witness exists, then z is declared as being dependent on Y . Procedure **isAutomataDependent** first uses Algorithm 1 to construct a list P of all compatible pairs in A (line 4). Then for every pair $(s, s') \in P$, the algorithm checks using procedure **AreStatesColliding** (lines 1-2) whether there exists an assignment σ, σ' for which both $\delta(s, \sigma)$ and $\delta(s', \sigma')$ exist, $\sigma.Y = \sigma'.Y$ and $\sigma.\{z\} \neq \sigma'.\{z\}$. If so, z is not dependent on Y (line 7) and the algorithm returns *false*. Otherwise, afterchecking all the pairs, the algorithm returns *true*.

Algorithm 1 Find All Compatible States in NBA

Input NBA $A_\varphi = (\Sigma, Q, \delta, q_0, F)$ of φ .
Output Set $P \subseteq Q \times Q$ of all compatible state pairs in A_φ

```

1:  $P \leftarrow \emptyset$ ;  $C \leftarrow \{(q_0, q_0)\}$ 
2: while  $C \neq \emptyset$  do
3:   Let  $(s_i, s_j) \in C$ 
4:    $P \leftarrow P \cup \{(s_i, s_j)\}$ ;  $C \leftarrow C \setminus \{(s_i, s_j)\}$ 
5:   for  $(s'_i, s'_j) \in \text{out}(s_1) \times \text{out}(s_2)$  do
6:     if  $(s'_i, s'_j) \notin P \cup C$  and  $\exists \sigma \in 2^\Sigma$  s.t.  $s'_i \in \delta(s_i, \sigma) \wedge s'_j \in \delta(s_j, \sigma)$  then
7:        $C \leftarrow C \cup \{(s'_i, s'_j)\}$ 
8:     end if
9:   end for
10: end while
11: return  $P$ 

```

Algorithm 2 Check Dependency Based Automaton

Input NBA $A_\varphi = (\Sigma, Q, \delta, q_0, F)$ from φ , Candidate dependent variable z , Candidate dependency set Y .
Output Is z dependent on Y by Definition 3

```

1: procedure ARESTATECOLLIDING( $p, q$ )
2:   return  $\exists \sigma_p, \sigma_q \in 2^\Sigma$  s.t.  $\delta(p, \sigma_p) \neq \emptyset \wedge \delta(q, \sigma_q) \neq \emptyset \wedge \sigma_p.Y = \sigma_q.Y \wedge \sigma_p.\{z\} \neq \sigma_q.\{z\}$ 
3: end procedure
4:  $P \leftarrow \text{FindAllCompatibleStates}(A_\varphi)$ 
5: for  $(s_1, s_2) \in P$  do
6:   if  $\text{AreStateColliding}(s_1, s_2)$  then
7:     return False
8:   end if
9: end for
10: return True

```

Lemma 1. *Algorithm 2 returns True if and only if z is automata-dependent on Y in A_φ .*

Using the above algorithm to perform dependency check, it is easy to compute a maximal set of dependent variables (as explained earlier). Note that all the above algorithms run in time polynomial (in fact, quadratic) in size of the NBA.

Corollary 1. *Given NBA A_φ , a maximal dependent set of outputs can be computed in time polynomial in the size of A_φ .*

Note that if all output variables are dependent, then regardless of the order in which the outputs are considered, for every finite history of inputs, there is a unique value for each output that makes the specification true. Therefore, there is a unique winning strategy for the specification, assuming it is realizable.

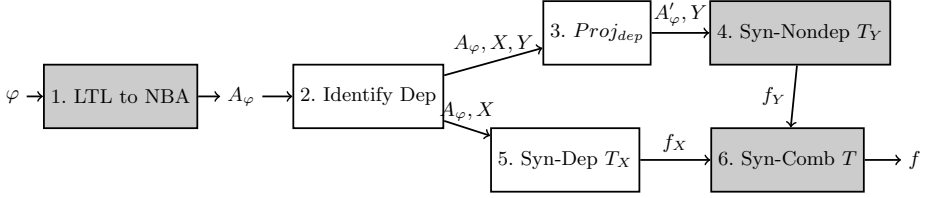


Fig. 2. Synthesis using dependencies. Note that Steps 2., 3., 5, are novel, while Steps 1., 4., 6. (shaded in gray) use pre-existing techniques.

4 Exploiting Dependency in Reactive Synthesis

In this section, we explain how dependencies can be beneficially exploited in a reactive synthesis pipeline. Our approach can be described at a high level as shown in Figure 2. This flow-chart has the following 6 steps:

1. Given an LTL formula φ over a set of variables V with input variables $I \subseteq V$ and output variables $O = V \setminus I$, we first construct a language-equivalent NBA $A_\varphi = (\Sigma_I \cup \Sigma_O, S, s_0, \delta, F)$ by standard means, e.g [29].
2. Then, as described in Section 3, we find in A_φ a maximal set of output variables X that are dependent in φ . For notational convenience, in the remainder of the discussion, we use Y for $I \cup (O \setminus X)$ and Σ_Y for $\Sigma_I \times \Sigma_{O \setminus X}$.
3. Next, we construct an NBA A'_φ from A_φ by projecting out (or eliminating) all X variables from labels of transitions. Thus, A'_φ has the same sets of states and transitions as A_φ . We simply remove valuations of variables in X from the label of every state transition in A_φ to obtain A'_φ . Note that after this step, $L(A'_\varphi) = \{w \mid \exists u \in L(A_\varphi) \text{ s.t. } w = u.Y\} \subseteq \Sigma_Y^*$.
4. Treating A'_φ as a (automata-based) specification with inputs I and outputs $O \setminus X$, we next use existing reactive synthesis techniques (e.g., [8]) to obtain a transducer T_Y that describes a strategy $f_Y : \Sigma_I^* \rightarrow \Sigma_{O \setminus X}$ for $L(A'_\varphi)$.
5. We also construct a transducer T_X that describes a function $f_X : (\Sigma_Y^* \rightarrow \Sigma_X)$ with the following property: for every word $w' \in L(A'_\varphi)$ there exists a unique word $w \in L(\varphi)$ such that $w.Y = w'$ and for all i , $w_i.X = f_X(w'[0, i])$.
6. Finally, we compose T_X and T_Y to construct a transducer T that defines the final strategy $f : \Sigma_I^* \rightarrow \Sigma_O$. Recall that transducer T_Y has I as inputs and $O \setminus X$ as outputs, while transducer T_X has I and $O \setminus X$ as inputs and X as outputs. Composing T_X and T_Y is done by simply connecting the outputs $O \setminus X$ of T_Y to the corresponding inputs of T_X .

In the above flow, we use standard techniques from the literature for Steps 1 and 4, as explained above. Hence we do not dwell on these steps in detail. Step 2 was detailed in Section 3. Step 3 is easy when we have an explicit representation of the automata, but it has interesting consequences when using symbolic representations of automata. Step 6 is also easy to implement. Hence, in the remainder of this section, we focus on Step 5, a key contribution of this paper. In the next section, we will discuss how steps 2, 3 and 5 are implemented using symbolic representations (viz. ROBDDs).

Constructing transducer T_X Let $A = (\Sigma_I \times \Sigma_O, Q, \delta, q_0, F)$ be the NBA A_φ obtained in step 1 of the pipeline shown above. Since each letter in Σ_O can be thought of as a pair (σ, σ') , where $\sigma \in \Sigma_{O \setminus X}$ and $\sigma' \in \Sigma_X$, the transition function δ can be viewed as a map from $Q \times (\Sigma_I \times \Sigma_{O \setminus X} \times \Sigma_X)$ to 2^Q . The transducer T_X we wish to construct is a deterministic Mealy machine described by the 6-tuple $(\Sigma_Y, \Sigma_X \cup \{\perp\}, Q^X, q_0^X, \delta^X, \lambda^X)$, where $\Sigma_Y = \Sigma_I \times \Sigma_{(O \setminus X)}$ is the input alphabet, Σ_X is the output alphabet with $\perp \notin \Sigma_X$ being a special symbol that is output when no symbol of Σ_X suffices, $Q^X = 2^Q$, that is the powerset of Q is the set of states of T_X , $q_0^X = \{q_0\}$ is the initial state, $\delta^X : Q^X \times \Sigma_I \times \Sigma_{(O \setminus X)} \rightarrow Q^X$ is the state transition function, and $\lambda^X : Q^X \times \Sigma_I \times \Sigma_{(O \setminus X)} \rightarrow \Sigma_X$ is the output function. The state transition function δ^X is defined by the Rabin-Scott subset construction applied to the automaton A_φ [19]. Formally, for every $U \subseteq Q$, $\sigma_I \in \Sigma_I$ and $\sigma \in \Sigma_{(O \setminus X)}$, we define $\delta^X(U, (\sigma_I, \sigma)) = \{q' \mid q' \in Q, \exists q \in U \text{ and } \exists \sigma' \in \Sigma_X \text{ s.t. } q' \in \delta(q, (\sigma_I, \sigma, \sigma'))\}$. Before defining the output function λ^X , we state an important property of T^X that follows from the definition of δ^X above.

Lemma 2. *If X is automata dependent in A_φ , then every state U reachable from q_0^X in T_X satisfies the property: $\forall q, q' \in U, (q, q')$ is compatible in A_φ .*

The lemma is easily proved by induction on the number of steps needed to reach U from q_0^X . Details of the proof may be found in [2]. We are now ready to define the output function λ^X of T_X . Let U be a state reachable from q_0^X in T_X and let $U' = \delta^X(U, (\sigma_I, \sigma))$, where $(\sigma_I, \sigma) \in \Sigma_Y$. If $U' \neq \emptyset$, we can infer that (see Proof of Lemma 2 in [2]) that there is a unique $\sigma_X \in \Sigma_X$ s.t. $U' = \{q' \mid \exists q \in U \text{ s.t. } q' \in \delta(q, (\sigma_I, \sigma, \sigma_X))\}$. We define $\lambda^X(U, (\sigma_I, \sigma)) = \sigma_X$ in this case. If, on the other hand, $U' = \emptyset$, we define $\lambda^X(U, (\sigma_I, \sigma)) = \perp$.

Theorem 3. *If φ is realizable, the transducer T obtained by composing T_X and T_Y as in step 6 of Fig. 2 solves the synthesis problem for φ .*

An interesting corollary of the above result is that for realizable specifications with all output variables dependent, we can solve the synthesis problem in time $O(2^k)$ instead of $\Omega(2^{k \log k})$, where $k = |A_\varphi|$. This is because the subset construction on A_φ suffices to obtain T_X , while A_φ must be converted to a deterministic parity automaton to solve the synthesis problem in general.

5 Symbolic Implementation

In this section, we describe symbolic implementations of each of the non-shaded steps in the synthesis flow depicted in Fig. 2. Before we delve into the details, a note on the representation of NBAs is relevant. We use the same representation as used in **Spot** [7] – a state-of-the-art platform for representing and manipulating LTL formulas and ω -automata. Specifically, the transition structure of an NBA A is represented as a directed graph, with nodes representing states of A , and directed edges representing state transitions. Furthermore, every edge from state s to state s' is labeled by a Boolean function $B_{(s, s')}$ over $I \cup O$. The Boolean

function can itself be represented in several forms. We assume it is represented as a Reduced Ordered Binary Decision Diagram (ROBDD) [10], as is done in **Spot**. Each such labeled edge represents a set of state transitions from s to s' , with one transition for each satisfying assignment of $B_{(s,s')}$.

Implementing Algorithms 1 and 2 (Step 2) : Since states of the NBA A_φ are explicitly represented as nodes of a graph, it is straightforward to implement Algorithms 1 and 2. The check in line 6 of Algorithm 1 is implemented by checking the satisfiability of $B_{(s_i,s'_i)}(I, O) \wedge B_{(s_j,s'_j)}(I, O)$ using ROBDD operations. Similarly, the check in line 2 of Algorithm 2 is implemented by checking the satisfiability of $\bigvee_{(s,s') \in \text{out}(p) \times \text{out}(q)} B_{(p,s)}(I, O) \wedge B_{(q,s')}(I', O') \wedge \bigwedge_{y \in Y} (y \leftrightarrow y') \wedge (z \leftrightarrow \neg z')$ using ROBDD operations. In the above formula, I' (resp. O') denotes a set of fresh, primed copies of variables in I (resp. O).

Implementing transformation of A_φ to A'_φ (Step 3): To obtain A'_φ , we simply replace the ROBDD for $B_{(s,s')}$ on every edge (s, s') of the NBA A_φ by an ROBDD for $\exists X B_{(s,s')}$. While the worst-case complexity of computing $\exists X B_{(s,s')}$ using ROBDDs is exponential in $|X|$, this doesn't lead to inefficiencies in practice because $|X|$ is typically small. Indeed, our experiments reveal that the total size of ROBDDs in the representation of A'_φ is invariably smaller, sometimes significantly, compared to the total size of ROBDDs in the representation of A_φ . Indeed, this reduction can be significant in some cases, as the following proposition shows (see proof in [2]).

Proposition 1. *There exists an NBA A_φ with a single dependent output such that the ROBDD labeling its edge is exponentially (in number of inputs and outputs) larger than that labeling the edge of A'_φ .*

Implementing transducer T_X (Step 5): We now describe how to construct a Mealy machine corresponding to the transducer T_X . As explained in the previous section, the transition structure of the Mealy machine is obtained by applying the subset construction to A_φ . While this requires $O(2^{|A_\varphi|})$ time if states and transitions are explicitly represented, we show below that a sequential circuit implementing the Mealy machine can be constructed directly from A_φ in time polynomial in $|X|$ and $|A_\varphi|$. This reduction in construction complexity crucially relies on the fact that all variables in X are dependent on $I \cup (O \setminus X)$.

Let $S = \{s_0, \dots, s_{k-1}\}$ be the set of states of A_φ , and let $\text{in}(s_i)$ denote the set of states that have an outgoing transition to s_i in A_φ . To implement the desired Mealy machine, we construct a sequential circuit with k state-holding flip-flops.

every state $U (\subseteq S)$ of the Mealy machine is represented by the state of these k flip-flops, i.e. by a k -dimensional Boolean vector. Specifically, the i^{th} component is set to 1 iff $s_i \in U$. For example, if $S = \{s_0, s_1, s_2\}$ and $U = \{s_0, s_2\}$, then U is represented by the vector $\langle 1, 0, 1 \rangle$. Let n_i and p_i denote the next-state input and present-state output of the i^{th} flip-flop. The next-state function δ^X from p'_i 's to n'_i 's of the Mealy machine is implemented by a circuit, say Δ^X , with inputs $\{p_0, \dots, p_{k-1}\} \cup I \cup (O \setminus X)$ and outputs $\{n_0, \dots, n_{k-1}\}$. For $i \in \{0, \dots, k -$

1}, output n_i of this circuit implements the Boolean function $\bigvee_{s_j \in \text{in}(s_i)} (p_j \wedge \exists X B_{(s_j, s_i)})$. To see why this works, suppose $\langle p_0, \dots, p_{k-1} \rangle$ represents the current state $U \subseteq S$ of the Mealy machine. Then the above function sets n_i to true iff there is a state $s_j \in U$ (i.e. $p_j = 1$) s.t. there is a transition from s_j to s_i on some values of outputs X and for the given values of $I \cup (O \setminus X)$ (i.e. $\exists X B_{(s_j, s_i)} = 1$). This is exactly the condition for s_i to be present in the state $U' \subseteq S$ reached from U for the given values of $I \cup (O \setminus X)$ in the Mealy machine obtained by subset construction.

It is known from the knowledge compilation literature (see e.g. [1, 4, 13]) that every ROBDD can be compiled in linear time to a Boolean circuit in Decomposable Negation Normal Form (DNNF), and that every DNNF circuit admits linear time projection of variables, yielding a resultant DNNF circuit. Hence, a Boolean circuit for $\exists X B_{(s_j, s_i)}$ can be constructed in time linear in the size of the ROBDD representation of $B_{(s_j, s_i)}$. This allows us to construct the circuit Δ^X , implementing the next-state transition logic of our Mealy machine, in time (and space) linear in $|X|$ and $|A_\varphi|$.

Next, we turn to constructing a circuit Λ^X that implements the output function λ^X of our Mealy machine. It is clear that Λ^X must have inputs $\{p_0, \dots, p_{k-1}\} \cup I \cup (O \setminus X)$ and outputs X . Since X is automata dependent on $I \cup (O \setminus X)$ in A_φ , the following proposition is easily seen to hold.

Proposition 2. *Let $B_{(s, s')}$ be a Boolean function with support $I \cup O$ that labels a transition (s, s') in A_φ . For every $(\sigma_I, \sigma) \in \Sigma_I \times \Sigma_{O \setminus X}$, if $(\sigma_I, \sigma) \models \exists X B_{(s, s')}$, then there is a unique $\sigma' \in \Sigma_X$ such that $(\sigma_I, \sigma, \sigma') \models B_{(s, s')}$.*

Considering only the transition (s, s') referred to in Proposition 2, we first discuss how to synthesize a vector of Boolean functions, say $F^{(s, s')} = \langle F_1^{(s, s')}, \dots, F_{|X|}^{(s, s')} \rangle$, where each component function has support $I \cup (O \setminus X)$, such that $F^{(s, s')}[I \mapsto \sigma_I][O \setminus X \mapsto \sigma] = \sigma'$. Generalizing beyond the specific assignment of $I \cup O$, our task effectively reduces to synthesizing an $|X|$ -dimensional vector of Boolean functions $F^{(s, s')}$ s.t. $\forall I \cup (O \setminus X) (\exists X B_{(s, s')} \rightarrow B_{(s, s')}[X \mapsto F^{(s, s')}])$ holds. Interestingly, this is an instance of *Boolean functional synthesis* – a problem that has been extensively studied in the recent past (see e.g. [1, 3, 4, 6, 11]). In fact, we know from [1, 26] that if $B_{(s, s')}$ is represented as an ROBDD, then a Boolean circuit for $F_{(s, s')}$ can be constructed in $\mathcal{O}(|X|^2 \cdot |B_{(s, s')}|)$ time, where $|B_{(s, s')}|$ denotes the size of the ROBDD for $B_{(s, s')}$. For every $x_i \in X$, we use this technique to construct a Boolean circuit for $F_i^{(s, s')}$ for every edge (s, s') in A . The overall circuit Λ^X is constructed such that the output for $x_i \in X$ implements the function $\bigvee_{\text{transition } (s, s') \text{ in } A} (p_s \wedge (B_{(s, s')}[X \mapsto F^{(s, s')}]) \wedge F_i^{(s, s')})$.

Lemma 3. *Let $U \subseteq S$ be a non-empty set of pairwise compatible states of A . For $(\sigma_I, \sigma) \in \Sigma_I \times \Sigma_{O \setminus X}$, if $\delta^X(U, (\sigma_I, \sigma)) \neq \emptyset$, then the outputs X of Λ^X evaluate to $\lambda^X(U, (\sigma_I, \sigma))$. In all other cases, every output of Λ^X evaluates to 0.*

Note that $\delta^X(U, (\sigma_I, \sigma)) = \emptyset$ iff all outputs n_i of the circuit Δ^X evaluate to 0. This case can be easily detected by checking if $\bigvee_{i=0}^{k-1} n_i$ evaluates to 0. We therefore have the following result.

Theorem 4. *The sequential circuit obtained with Δ^X as next-state function and A^X as output function is a correct implementation of transducer T_X , assuming (a) the initial state is $p_0 = 1$ and $p_j = 0$ for all $j \in \{1, \dots, k-1\}$, and (b) the output is interpreted as \perp whenever $\bigvee_{i=0}^{k-1} n_i$ evaluates to 0.*

6 Experiments and Evaluation

We implemented the synthesis pipeline depicted in Figure 2 in a tool called DepSynt (accessible at <https://github.com/eliyao32/DepSynt>), using symbolic approach of Section 5. For Steps 1., 4., of the pipeline, i.e., construction of A_φ and synthesis of T_Y , we used the tool Spot [7], a widely used library for representing and manipulating NBAs. We then experimented with all available reactive synthesis benchmarks from the SYNTCOMP [21] competition, a total of 1,141 LTL specifications over 31 benchmark families.

All our experiments were run on a computer cluster, with each problem instance run on an Intel Xeon Gold 6130 CPU clocking at 2.1 GHz with 2GB memory and running Rocky Linux 8.6. Our investigation was focussed on answering two main research questions:

RQ1: How prevalent are dependent outputs in reactive synthesis benchmarks?

RQ2: Under what conditions, if any, is reactive synthesis benefited by our approach, i.e., of identifying and separately processing dependent output variables?

Dependency Prevalence. To answer **RQ1**, we implemented the algorithm in Section 3 and executed it with a timeout of 1 hour. Within this time, we were able to find 300 benchmarks out of 1,141 SYNTCOMP benchmarks, that had at least 1 dependent output variable (as per Definition 3). Out of the 1,141 benchmarks, 260 had either timeout (41 total) or out-of-memory (219 total), out of which 227 failed because of the NBA construction (adapted from Spot), i.e, Step 1 in our pipeline, did not terminate. We found that all the benchmarks with at least 1 dependent variable in fact belong to one of 5 benchmark families, as seen in Table 1. In order to measure the prevalence of dependency we evaluated (1) the number of dependent variables and (2) the dependency ratio = $\frac{\text{Total dependent vars}}{\text{Total output vars}}$. Out of those depicted, Mux (for mul-

Benchmark Family	Total	Completed	Found Dep	Avg Dep Ratio
ltl2dpa	24	24	24	.434
mux	12	12	4	1
shift	11	4	4	1
tsl-paper	118	117	115	.46
tsl-smart-home-jarvis	189	167	153	.33

Table 1. Summary for 5 benchmark families, indicating the no. of benchmarks, where the dependency-finding process was completed, the total count of benchmarks with dependent variables, and the average dependency ratio among those with dependencies.

tiplexer) and shift (for shift-operator operator) were two benchmark families

where dependency ratio was 1. In total, among all those where our dependency checking algorithm terminated, we found 26 benchmarks with all the output variables dependent. Of these 4 benchmarks were from Shift, 4 benchmarks from mux, 14 benchmarks from tsl-paper, and 4 from tsl-smart-home-jarvis. Looking beyond total dependency, among the 300 benchmarks with at least 1 dependent variable, we found a diverse distribution of dependent variables as shown in Figure 3 (distribution wrt dependency ratio is in [2]).

Utilizing Dependency for Reactive Synthesis: Comparison with other tools.

Despite a large 1 hr time out, we noticed that most dependent variables were found within 10-12 seconds. Hence, in our tool DepSynt, we limited the time for dependency-check to an empirically determined 12 seconds, and declared unchecked variables after this time as non-dependent. Since synthesis of non-dependents T_Y (Step 5. of the pipeline) is implemented directly using Spot APIs, the difference between our approach and Spot is minimal when there are a large number of non-dependent variables. This motivated us to divide our experimental comparison, among the 300 benchmarks where at least one dependent variables was found, into benchmarks with at most 3 non-dependent variables (162 benchmarks) and more than 3 non-dependent variables (138 benchmarks). We compared DepSynt with two state-of-the-art synthesis tools, that won in different tracks of SYNTCOMP23' [21]: (i) Ltlsynt (based on Spot) [7] with different configurations ACD, SD, DS, LAR, and (ii) Strix [22] with the configuration of BFS for exploration and FPI as parity game solver (the overall winning configuration/tool in SYNTCOMP'23). All the tools had a total timeout of 3 hours per benchmark. As can be seen from Figure 4, indeed for the case of ≤ 3 non-dependent variables, DepSynt outperforms the highly optimized competition-winning tools. Even for > 3 case, as shown in Figure 5, the performance of DepSynt is comparable to other tools, only beaten eventually by Strix. DepSynt uniquely solved 2 specifications for which both Strix and Ltlsynt timed out after 3600s, the benchmarks are mux32, and mux64, and solved in 2ms, and 4ms respectively.

Analyzing time taken by different parts of the pipeline. In order to better understand where DepSynt spends its time, we plotted in Figure 6 the normalized time distribution of DepSynt. We can see that synthesizing a strategy for dependent variables is very fast (the yellow portion)- justifying its theoretical linear complexity bound, and so is the pink region depicting searching for dependency

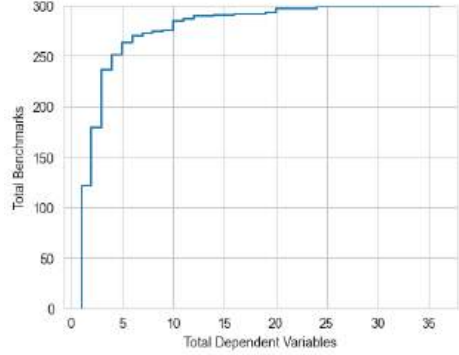


Fig. 3. Cumulative count of benchmarks for each unique value of Total Dependent Variables. $F(x)$ on y-axis represents how many benchmarks have at most x (on x-axis) dependent variables.

(again, a poly-time algorithm), especially compared to the blue synthesizing a strategy for the non-dependent variables, and the green which is NBA build time. This also explains why having a high dependency ratio alone does not help our approach, since even with a high ratio, the number of non-dependent variables could be large, resulting in worse performance overall.

Analysis of the Projection step (Step 3.) of Pipeline. The rationale for projecting variables from the NBA is to reduce the number of output non-dependent variables in the synthesis of the NBA, which is the most expensive phase as Figure 6 shows. To see if this indeed contributes to our better performance, we asked if projecting the dependent variables reduces the BDDs’ sizes, in terms of total nodes, (the BDD represents the transitions). Figure 7 shows that the BDDs’ sizes are reduced significantly where the total of non-dependent variables is at most 3, in cases of total dependency, the BDD just vanishes and is replaced by the constant true/false. For the case of total non-dependent is 4 or more, the BDD size is reduced as well.

An ablation experiment with Spot. As a final check, that dependency was causing the improvements seen, we conducted a control/ablation experiment where in DepSynt we gave zero-timeout to find dependency, classified all output variables as non-dependent, and called this SpotModular. As can be seen in Figure 8, for the case of benchmarks with at least 1 dependent and at most 3 non-dependent variables, this clearly shows the benefit of dependency-checking. In the full version [2], we show that for other cases we do not see this.

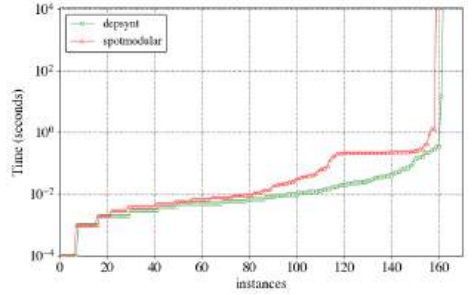


Fig. 8. Cactus plot comparing DepSynt and SpotModular on 162 benchmarks with at most 3 non-dependent variables.

As can be seen in Figure 8, for the case of benchmarks with at least 1 dependent and at most 3 non-dependent variables, this clearly shows the benefit of dependency-checking. In the full version [2], we show that for other cases we do not see this.

Summary. Overall, we answered both the research questions we started with. Indeed there are several benchmarks with dependent variables, and using our pipeline does give performance benefits when no. of non-dependent variables is low. Our recipe would be to first run our poly-time check to see if there are dependents and use our approach if there are not too many non-dependents; otherwise switch to any existing method. To summarize our comparisons: wrt Strix, we found 252 benchmarks that had dependent variables in which DepSynt took less time than Strix. Out of which, in 126 benchmarks DepSynt took at least 1 second less than Strix. Among these, for 10 benchmarks (shift16, LightsTotal.d65ed84e, LightsTotal.9cbf2546, LightsTotal.06e9cad4, Lights2.f3987563, Lights2.0f5381e9, FelixSpecFixed3.core_b209ff21, Lights2.b02056d6, Lights2.06e9cad4, LightsTotal.2c5b09da) the time taken by DepSynt was at least 10 seconds less than that taken by Strix. These are the examples that are easier to solve by DepSynt than by Strix. For shift16, the difference was more than 1056 seconds in favor of DepSynt. Interestingly, shift16 also has all output variables dependent.

When comparing with Ltlsynt, we found 193 benchmarks that had dependent variables in which DepSynt took less time than Ltlsynt. Among these, in 27 benchmarks DepSynt took at least 1 second less than Ltlsynt. Of these, there is one benchmark (ModifiedLedMatrix5X) for which the time taken by DepSynt was at least 10 seconds less than that taken by Ltlsynt. Specifically, DepSynt took 5 seconds and Ltlsynt took 55 seconds.

7 Conclusion

In this work, we have introduced the notion of dependent variables in the context of reactive synthesis. We showed that dependent variables are prevalent in reactive synthesis benchmarks and suggested a synthesis approach that may utilize these dependency for better synthesis. As part of future work, we wish to explore heuristics for choosing "good" maximal subsets of dependent variables. We also wish to explore integration of our method in other reactive synthesis tools such as Strix.

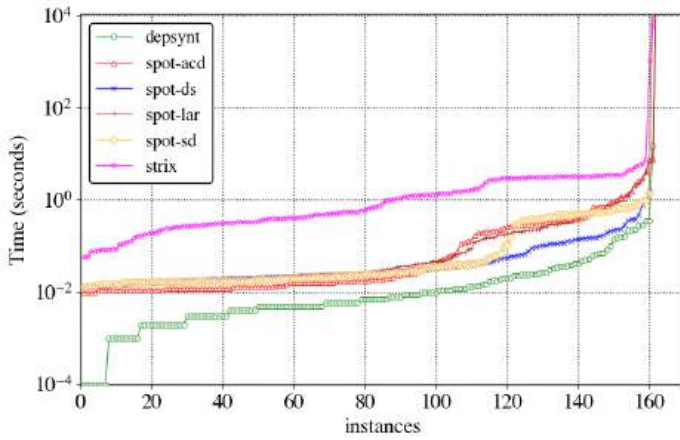


Fig. 4. Cactus plot comparing DepSynt, LtlSynt, and Strix on 162 benchmarks with at most 3 non-dependent variables.

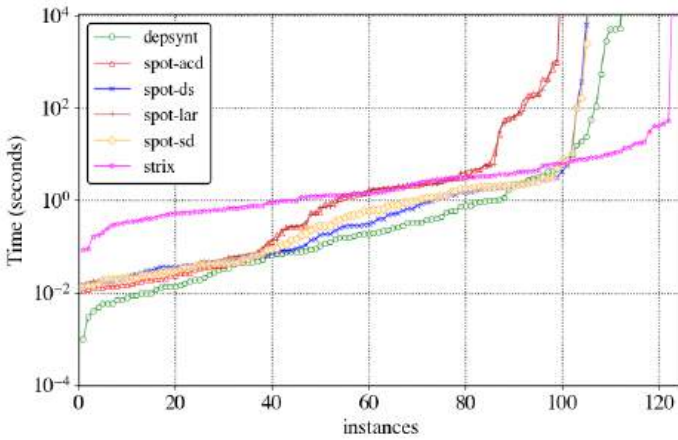


Fig. 5. Cactus plot comparing DepSynt, LtlSynt, and Strix on 138 benchmarks with more than 3 non-dependent variables.

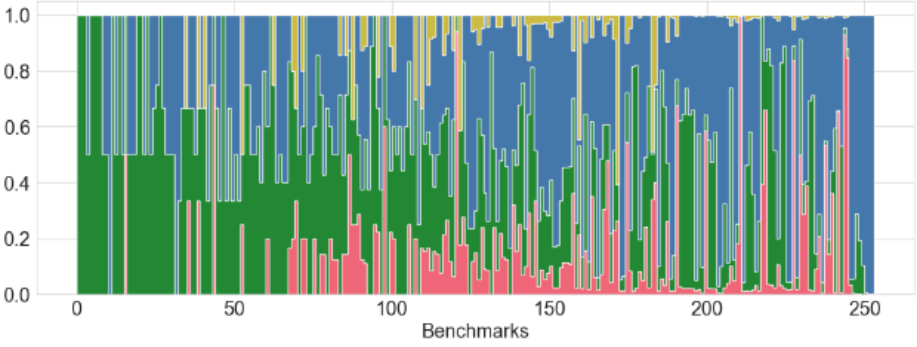


Fig. 6. Normalized time distribution of DepSynt sorted by total duration over benchmarks that could be solved successfully by DepSynt. Each color represents a different phase of DepSynt. Pink is searching for dependency, green is the NBA build, blue is synthesis of non-dependent variables and yellow is dependent variables synthesis.

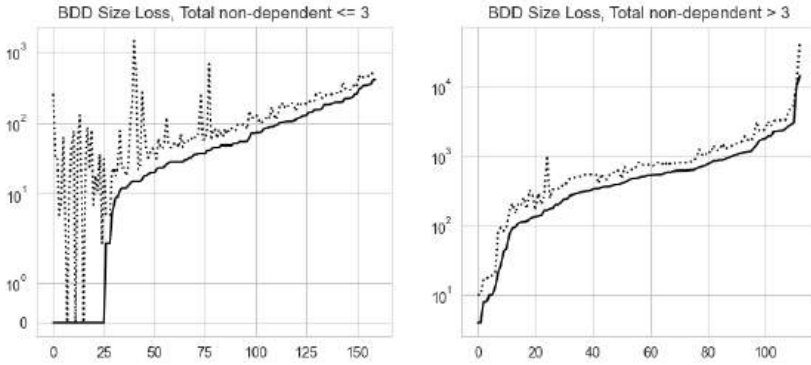


Fig. 7. This figure illustrates the total BDD sizes of the NBA edges before and after the projection of the dependent variables from the NBA edges, the left figure is over benchmarks with at most 3 non-dependent variables and the right figure is over benchmarks with 4 or more non-dependent variables. The solid line presents the projected BDD size and the dotted line presents the original BDD size. The y-axis is presented in symmetric log-scale. Benchmarks are sorted by the projected NBA's BDD total size.

References

1. Akshay, S., Arora, J., Chakraborty, S., Krishna, S.N., Raghunathan, D., Shah, S.: Knowledge compilation for boolean functional synthesis. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22–25, 2019. pp. 161–169. IEEE (2019)
2. Akshay, S., Basa, E., Chakraborty, S., Fried, D.: On dependent variables in reactive synthesis (full version). arXiv preprint arXiv:2401.11290 (2024)
3. Akshay, S., Chakraborty, S.: Synthesizing skolem functions: A view from theory and practice. In: Sarukkai, S., Chakraborty, M. (eds.) Handbook of Logical Thought in India, pp. 1–36. Springer (2022)
4. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What’s hard about boolean functional synthesis? 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. Lecture Notes in Computer Science, vol. 10981, pp. 251–269. Springer (2018)
5. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Boolean functional synthesis: hardness and practical algorithms. *Formal Methods Syst. Des.* **57**(1), 53–86 (2021). <https://doi.org/10.1007/s10703-020-00352-2>, <https://doi.org/10.1007/s10703-020-00352-2>
6. Amram, G., Bansal, S., Fried, D., Tabajara, L.M., Vardi, M.Y., Weiss, G.: Adapting behaviors via reactive synthesis. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 870–893. Springer (2021)
7. Blahoudek, F., Duret-Lutz, A., Strejček, J.: Seminotor 2 can complement generalized Büchi automata via improved semi-determinization. In: Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV’20). Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (Jul 2020)
8. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 921–962. Springer (2018)
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa’ar, Y.: Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.* **78**(3), 911–938 (2012)
10. Bryant, R.E.: Binary decision diagrams and beyond: Enabling technologies for formal verification. In: Proceedings of IEEE International Conference on Computer Aided Design (ICCAD). pp. 236–243. IEEE (1995)
11. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. *Formal Methods Syst. Des.* **60**(2), 228–258 (2022)
12. Church, A.: Logic, arithmetic, and automata. In: International Congress of Mathematicians. p. 23–35 (1962)
13. Darwiche, A.: Decomposable negation normal form. *J. ACM* **48**(4), 608–647 (2001)
14. Faymonville, P., Finkbeiner, B., Tentrup, L.: Bony: An experimentation framework for bounded synthesis. In: Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017, Proceedings, Part II. pp. 325–332. Springer (2017)
15. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12673, pp. 113–130. Springer (2021)

16. Finkbeiner, B., Schewe, S.: Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.* **15**(5-6), 519–539 (2013)
17. Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for boolean function synthesis. *Computer Aided Verification* **12225**, 611 – 633 (2020)
18. Golia, P., Slivovsky, F., Roy, S., Meel, K.S.: Engineering an efficient boolean functional synthesis engine. *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* pp. 1–9 (2021)
19. Hopcroft, J.E., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company (1979)
20. Huth, M., Ryan, M.: *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, USA (2004)
21. Jacobs, S., Perez, G.A., Abraham, R., Bruyere, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (syntcomp): 2018–2021 (2022)
22. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! 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*. pp. 578–586. Springer (2018)
23. Michaud, T., Colange, M.: Reactive synthesis from ltl specification with spot. In: *Proceedings of the 7th Workshop on Synthesis, SYNT@ CAV* (2018)
24. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 179–190 (1989)
25. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings*. pp. 375–392 (2016)
26. Shah, P., Bansal, A., Akshay, S., Chakraborty, S.: A normal form characterization for efficient boolean skolem function synthesis. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. pp. 1–13. IEEE (2021). <https://doi.org/10.1109/LICS52264.2021.9470741>, <https://doi.org/10.1109/LICS52264.2021.9470741>
27. Soos, M., Meel, K.S.: Arjun: An efficient independent support computation technique and its applications to counting and sampling. In: *ICCAD* (Nov 2022)
28. Tseitin, G.S.: On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* pp. 466–483 (1983)
29. Vardi, M., Wolper, P.: Reasoning about infinite computations. *Information and Computation* **115**(1), 1–37 (1994). <https://doi.org/10.1006/inco.1994.1092> <https://www.sciencedirect.com/science/article/pii/S0890540184710923>

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.





CESAR: Control Envelope Synthesis via Angelic Refinements ^{*}

Aditi Kabra^{1(✉)} , Jonathan Laurent^{1,2} , Stefan Mitsch^{1,3} ,
and André Platzer^{1,2}

¹ Carnegie Mellon University, Pittsburgh, USA
akabra@cs.cmu.edu

² Karlsruhe Institute of Technology, Karlsruhe, Germany
{jonathan.laurent, platzer}@kit.edu

³ DePaul University, Chicago, USA
smitsch@depaul.edu

Abstract. This paper presents an approach for synthesizing provably correct control envelopes for hybrid systems. Control envelopes characterize families of safe controllers and are used to monitor untrusted controllers at runtime. Our algorithm fills in the blanks of a hybrid system’s sketch specifying the desired shape of the control envelope, the possible control actions, and the system’s differential equations. In order to maximize the flexibility of the control envelope, the synthesized conditions saying which control action can be chosen when should be as permissive as possible while establishing a desired safety condition from the available assumptions, which are augmented if needed. An implicit, optimal solution to this synthesis problem is characterized using hybrid systems game theory, from which explicit solutions can be derived via symbolic execution and sound, systematic game refinements. Optimality can be recovered in the face of approximation via a dual game characterization. The resulting algorithm, *Control Envelope Synthesis via Angelic Refinements (CESAR)*, is demonstrated in a range of safe control envelope synthesis examples with different control challenges.

Keywords: Hybrid systems · Program synthesis · Differential game logic

1 Introduction

Hybrid systems are important models of many applications, capturing their differential equations and control [27,41,3,33,4,28]. For overall system safety, the correctness of the control decisions in a hybrid system is crucial. Formal verification techniques can justify correctness properties. Such correct controllers have

^{*} This work was funded by the Federal Railroad Administration Office of Research, Development and Technology under contract number 693JJ620C000025, a Swartz Center Innovation Commercialization Fellowship, and an Alexander von Humboldt Professorship.

been identified in a sequence of challenging case studies [34,40,12,32,19,14,22]. A useful approach to verified control is to design and verify a safe *control envelope* around possible safe control actions. Safe control envelopes are nondeterministic programs whose every execution is safe. In contrast with controllers, control envelopes define entire families of controllers to allow control actions under as many circumstances as possible, as long as they maintain the safety of the hybrid system. Safe control envelopes allow the verification of abstractions of control systems, isolating the parts relevant to the safety feature of interest, without involving the full complexity of a specific control implementation. The full control system is then monitored for adherence to the safe control envelope at runtime [29]. The control envelope approach allows a single verification result to apply to multiple specialized control implementations, optimized for different objectives. It puts industrial controllers that are too complex to verify directly within the reach of verification, because a control envelope only needs to model the safety-critical aspects of the controller. Control envelopes also enable applications like justified speculative control [17], where machine-learning-based agents control safety-critical systems safeguarded within a verified control envelope, or [36], where these envelopes generate reward signals for reinforcement learning.

Control envelope design is challenging. Engineers are good at specifying the *shape* of a model and listing the possible control actions by translating client specifications, which is crucial for the fidelity of the resulting model. But identifying the exact control conditions required for safety in a model is a much harder problem that requires design insights and creativity, and is the main point of the deep area of control theory. Most initial system designs are incorrect and need to be fixed before verification succeeds. Fully rigorous justification of the safety of the control conditions requires full verification of the resulting controller in the hybrid systems model. We present a synthesis technique that addresses this hard problem by filling in the holes of a hybrid systems model to identify a correct-by-construction control *envelope* that is as permissive as possible.

Our approach is called *Control Envelope Synthesis via Angelic Refinements (CESAR)*. The idea is to implicitly characterize the optimal safe control envelope via hybrid games yielding maximally permissive safe solutions in differential game logic [33]. To derive explicit solutions used for controller monitoring at runtime, we successively refine the games while preserving safety and, if possible, optimality. Our experiments demonstrate that CESAR solves hybrid systems synthesis challenges requiring different control insights.

Contributions. The primary contributions of this paper behind CESAR are:

- optimal hybrid systems control envelope synthesis via hybrid games.
- differential game logic formulas identifying optimal safe control envelopes.
- refinement techniques for safe control envelope approximation, including *bounded fixpoint unrollings* via a recurrence, which exploits *action permanence* (a hybrid analogue to idempotence).
- a primal/dual game counterpart optimality criterion.

2 Background: Differential Game Logic

We use hybrid games written in differential game logic (dGL, [33]) to represent solutions to the synthesis problem. Hybrid games are two-player noncooperative zero-sum sequential games with no draws that are played on a hybrid system with differential equations. Players take turns and in their turn can choose to act arbitrarily within the game rules. At the end of the game, one player wins, the other one loses. The players are classically called Angel and Demon. *Hybrid systems*, in contrast, have no agents, only a nondeterministic controller running in a nondeterministic environment. The synthesis problem consists of filling in holes in a hybrid system. Thus, expressing solutions for hybrid *system* synthesis with hybrid *games* is one of the insights of this paper.

An example of a game is $(v := 1 \sqcap v := -1); \{x' = v\}$. In this game, first Demon chooses between setting velocity v to 1, or to -1. Then, Angel evolves position x as $x' = v$ for a duration of her choice. Differential game logic uses modalities to set win conditions for the players. For example, in the formula $[(v := 1 \sqcap v := -1); \{x' = v\}] x \neq 0$, Demon wins the game when $x \neq 0$ at the end of the game and Angel wins otherwise. The overall formula represents the set of states from which Demon can win the game, which is $x \neq 0$ because when $x < 0$, Demon has the *winning strategy* to pick $v := -1$, so no matter how long Angel evolves $x' = v$, x remains negative. Likewise, when $x > 0$, Demon can pick $v := 1$. However, when $x = 0$, Angel has a winning strategy: to evolve $x' = v$ for zero time, so that x remains zero regardless of Demon's choice.

We summarize dGL's program notation (Table 1). See [33] for full exposition. Assignment $x := \theta$ instantly changes the value of variable x to the value of θ . Challenge $? \psi$ continues the game if ψ is satisfied in the current state, otherwise Angel loses immediately. In continuous evolution $x' = \theta \ \& \ \psi$ Angel follows the differential equation $x' = \theta$ for some duration of her choice, but loses immediately on violating ψ at any time. Sequential game $\alpha; \beta$ first plays α and when it

Table 1: Hybrid game operators for two-player hybrid systems

Game	Effect
$x := \theta$	assign value of term θ to variable x
$? \psi$	Angel passes challenge if formula ψ holds in current state, else loses immediately
$(x'_1 = \theta_1, \dots, x'_n = \theta_n \ \& \ \psi)$	Angel evolves x_i along differential equation system $x'_i = \theta_i$ for choice of duration ≥ 0 , loses immediately when violating ψ
$\alpha; \beta$	sequential game, first play hybrid game α , then hybrid game β
$\alpha \sqcup \beta$	Angel chooses to follow either hybrid game α or β
α^*	Angel repeats hybrid game α , choosing to stop or go after each α
α^d	dual game switches player roles between Angel and Demon
$\alpha \sqcap \beta$	demonic choice $(\alpha^d \sqcup \beta^d)^d$ gives choice between α and β to Demon
α^\times	demonic repetition $((\alpha^d)^*)^d$ gives control of repetition to Demon

terminates without a player having lost, continues with β . Choice $\alpha \cup \beta$ lets Angel choose whether to play α or β . For repetition α^* , Angel repeats α some number of times, choosing to continue or terminate after each round. The dual game α^d switches the roles of players. For example, in the game $?\psi^d$, Demon passes the challenge if the current state satisfies ψ , and otherwise loses immediately.

In games restricted to the structures listed above but without α^d , all choices are resolved by Angel alone with no adversary, and hybrid games coincide with hybrid systems in differential dynamic logic (dL) [33]. We will use this restriction to specify the synthesis *question*, the sketch that specifies the shape and safety properties of control envelopes. But to characterize the *solution* that fills in the blanks of the control envelope sketch, we use games where both Angel and Demon play. Notation we use includes demonic choice $\alpha \cap \beta$, which lets Demon choose whether to run α or β . Demonic repetition α^\times lets Demon choose whether to repeat α choosing whether to stop or go at the end of every run. We define $\alpha^{*\leq n}$ and $\alpha^{\times\leq n}$ for angelic and demonic repetitions respectively of at most n times.

In order to express properties about hybrid games, differential game logic formulas refer to the existence of winning strategies for objectives of the games (e.g., a controller has a winning strategy to achieve collision avoidance despite an adversarial environment). The set of dGL formulas is generated by the following grammar (where $\sim \in \{<, \leq, =, \geq, >\}$ and θ_1, θ_2 are arithmetic expressions in $+$, $-$, \cdot , $/$ over the reals, x is a variable, α is a hybrid game):

$$\phi := \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall x \phi \mid \exists x \phi \mid [\alpha] \phi \mid \langle \alpha \rangle \phi$$

Comparisons of arithmetic expressions, Boolean connectives, and quantifiers over the reals are as usual. The modal formula $\langle \alpha \rangle \phi$ expresses that player Angel has a winning strategy to reach a state satisfying ϕ in hybrid game α . Modal formula $[\alpha] \phi$ expresses the same for Demon. The fragment without modalities is first-order real arithmetic. Its fragment without quantifiers is called *propositional arithmetic* $\mathcal{P}_{\mathbb{R}}$. Details on the semantics of dGL can be found in [33]. A formula ϕ is *valid*, written $\models \phi$, iff it is true in every state ω . States are functions assigning a real number to each variable. For instance, $\phi \rightarrow [\alpha] \psi$ is valid iff, from all initial states satisfying ϕ , Demon has a winning strategy in game α to achieve ψ .

Control Safety Envelopes by Example. In order to separate safety critical aspects from other system goals during control design, we abstractly describe the safe choices of a controller with safe control envelopes that deliberately underspecify when and how to exactly execute certain actions. They focus on describing in which regions it is safe to take actions. For example, Model 1 designs a train control envelope [34] that must stop by the train by the *end of movement authority* e located somewhere ahead, as assigned by the train network scheduler. Past e , there may be obstacles or other trains. The train's control choices are to accelerate or brake as it moves along the track. The goal of CESAR is to synthesize the framed formulas in the model, that are initially blank.

Line 6 describes the *safety property* that is to be enforced at all times: the train driving at position p with velocity v must not go past position e . Line 1

Model 1 The train ETCS model (slightly modified from [34]). Framed formulas are initially blank and are automatically synthesized by our tool as indicated.

assum	1	$A > 0 \wedge B > 0 \wedge T > 0 \wedge v \geq 0 \wedge$
ctrlable	2	$\boxed{e - p > v^2/2B} \rightarrow \{ \{$
ctrl	3	$(\quad (? \boxed{e - p > vT + AT^2/2 + (v + AT)^2/2B} ; a := A)$
	4	$\cup (? \boxed{\text{true}} ; a := -B) \quad) ;$
plant	5	$(t := 0 ; \{p' = v, v' = a, t' = 1 \ \& \ t \leq T \wedge v \geq 0\})$
safe	6	$\}^*)(e - p > 0)$

lists *modeling assumptions*: the train is capable of both acceleration ($A > 0$) and deceleration ($B > 0$), the controller latency is positive ($T > 0$) and the train cannot move backwards as a product of braking (this last fact is also reflected by having $v \geq 0$ as a domain constraint for the plant on Line 5). These assumptions are fundamentally about the physics of the problem being considered. In contrast, Line 2 features a *controllability assumption* that can be derived from careful analysis. Here, this synthesized assumption says that the train cannot start so close to e that it won't stop in time even if it starts braking immediately. Line 3 and Line 4 describe a train controller with two actions: accelerating ($a := A$) and braking ($a := -B$). Each action is guarded by a synthesized formula, called an *action guard* that indicates when it is safe to use. Angel has control over which action runs, and adversarially plays with the objective of violating safety conditions. But Angel's options are limited to only safe ones because of the synthesized action guards, ensuring that Demon still wins and the overall formula is valid. In this case, braking is always safe whereas acceleration can only be allowed when the distance to end position e is sufficiently large. Finally, the plant on Line 5 uses differential equations to describe the train's kinematics. A timer variable t is used to ensure that no two consecutive runs of the controller are separated by more than time T . Thus, this controller is *time-triggered*.

Overview of CESAR. CESAR first identifies the optimal solution for the blank of Line 2. Intuitively, this blank should identify a *controllable invariant*, which denotes a set of states where a controller with choice between acceleration and braking has some strategy (to be enforced by the conditions of Line 3 and Line 4) that guarantees safe control forever. Such states can be characterized by the following dGL formula where Demon, as a proxy for the controller, decides whether to accelerate or brake: $(((a := A \wedge a := -B) ; \text{plant})^*) \text{safe}$ where *plant* and *safe* are from Model 1. When this formula is true, Demon, who decides when to brake to maintain the safety contract, has a winning strategy that the controller can mimic. When it is false, Demon, a perfect player striving to maintain safety, has no winning strategy, so a controller has no guaranteed way to stay safe either.

This dGL formula provides an *implicit* characterization of the optimal controllable invariant from which we derive an explicit formula in $\mathcal{P}_{\mathbb{R}}$ to fill the blank with using symbolic execution. Symbolic execution solves a game following the

axioms of dGL to produce an equivalent $\mathcal{P}_{\mathbb{R}}$ formula (Section 3.7). However, our dGL formula contains a loop, for which symbolic execution will not terminate in finite time. To reason about the loop, we *refine* the game, modifying it so that it is easier to symbolically execute, but still at least as hard for Demon to win so that the controllable invariant that it generates remains sound. In this example, the required game transformation first restricts Demon's options to braking. Then, it eliminates the loop using the observation that the repeated hybrid iterations $(a := -B; \text{plant})^*$ behave the same as just following the continuous dynamics of braking for unbounded time. It replaces the original game with $a := -B; t := 0; \{p' = v, v' = a \ \& \ \wedge v \geq 0\}$, which is loop-free and easily symbolically executed. Symbolically executing this game to reach safety condition **safe** yields controllable invariant $e - p > \frac{v^2}{2B}$ to fill the blank of Line 2.

Intuitively, this refinement (formalized in Section 3.4) captures situations where the controller stays safe forever by picking a single control action (braking). It generates the optimal solution for this example because braking forever is the dominant strategy: given any state, if braking forever does not keep the train safe, then certainly no other strategy will. However, there are other problems where the dominant control strategy requires the controller to strategically switch between actions, and this refinement misses some controllable invariant states. So we introduce a new refinement: bounded game unrolling via a recurrence (Section 3.5). A solution generated by unrolling n times captures states where the controller can stay safe by switching control actions up to n times.

Having synthesized the controllable invariant, CESAR fills the action guards (Line 3 and Line 4). An action should be permissible when running it for one iteration maintains the controllable invariant. For example, acceleration is safe to execute exactly when $[a := A; \text{plant}]e - p > \frac{v^2}{2B}$. We symbolically execute this game to synthesize the formula that fills the guard of Line 3.

3 Approach

This section formally introduces the *Control Envelope Synthesis via Angelic Refinements* (CESAR) approach for hybrid systems control envelope synthesis.

3.1 Problem Definition

We frame the problem of *control envelope synthesis* in terms of filling in holes \sqcup in a problem of the following shape:

$$\text{prob} \equiv \text{assum} \wedge \sqcup \rightarrow [((\cup_i (? \sqcup_i; \text{act}_i)); \text{plant})^*)] \text{safe}. \quad (1)$$

Here, the control envelope consists of a nondeterministic choice between a finite number of guarded actions. Each action act_i is guarded by a condition \sqcup_i to be determined in a way that ensures safety within a controllable invariant [6,18] \sqcup to be synthesized also. The plant is defined by the following template:

$$\text{plant} \equiv t := 0; \{x' = f(x), t' = 1 \ \& \ \text{domain} \wedge t \leq T\}. \quad (2)$$

This ensures that the plant must yield to the controller after time T at most, where T is assumed to be positive and constant. In addition, we make the following assumptions:

1. Components **assum**, **safe** and **domain** are propositional arithmetic formulas.
2. Timer variable t is fresh (does not occur except where shown in template).
3. Programs act_i are discrete **dL** programs that can involve choices, assignments and tests with propositional arithmetic. Variables assigned by act_i must not appear in **safe**. In addition, act_i must terminate in the sense that $\models \langle \text{act}_i \rangle \text{true}$.
4. The modeling assumptions **assum** are invariant in the sense that $\models \text{assum} \rightarrow [(\cup_i \text{act}_i) ; \text{plant}] \text{assum}$. This holds trivially for assumptions about constant parameters such as $A > 0$ in Model 1 and this ensures that the controller can always rely on them being true.

Definition 1. A solution to the synthesis problem above is defined as a pair (I, G) where I is a formula and G maps each action index i to a formula G_i . In addition, the following conditions must hold:

1. *Safety is guaranteed:* $\text{prob}(I, G) \equiv \text{prob}[\sqcup \mapsto I, \sqcup_i \mapsto G_i]$ is valid and $(\text{assum} \wedge I)$ is a loop invariant that proves it so.
2. *There is always some action:* $(\text{assum} \wedge I) \rightarrow \bigvee_i G_i$ is valid.

Condition 2 is crucial for using the resulting nondeterministic control envelope, since it guarantees that safe actions are always available as a fallback.

3.2 An Optimal Solution

Solutions to a synthesis problem may differ in quality. Intuitively, a solution is better than another if it allows for a strictly larger controllable invariant. In case of equality, the solution with the more permissive control envelope wins. Formally, given two solutions $S = (I, G)$ and $S' = (I', G')$, we say that S' is better or equal to S (written $S \sqsubseteq S'$) if and only if $\models \text{assum} \rightarrow (I \rightarrow I')$ and additionally either $\models \text{assum} \rightarrow \neg(I' \rightarrow I)$ or $\models (\text{assum} \wedge I) \rightarrow \bigwedge_i (G_i \rightarrow G'_i)$. Given two solutions S and S' , one can define a solution $S \sqcap S' = (I \vee I', i \mapsto (I \wedge G_i \vee I' \wedge G'_i))$ that is better or equal to both S and S' ($S \sqsubseteq S \sqcap S'$ and $S' \sqsubseteq S \sqcap S'$). A solution S' is called the *optimal solution* when it is the maximum element in the ordering, so that for any other solution S , $S \sqsubseteq S'$. The optimal solution exists and is expressible in **dGL**:

$$I^{\text{opt}} \equiv [((\cap_i \text{act}_i) ; \text{plant})^*] \text{safe} \quad (3)$$

$$G_i^{\text{opt}} \equiv [\text{act}_i ; \text{plant}] I^{\text{opt}}. \quad (4)$$

Intuitively, I^{opt} characterizes the set of all states from which an optimal controller (played here by Demon) can keep the system safe forever. In turn, G^{opt} is defined to allow any control action that is guaranteed to keep the system within I^{opt} until the next control cycle as characterized by a modal formula. Section 3.3 formally establishes the correctness and optimality of $S^{\text{opt}} \equiv (I^{\text{opt}}, G^{\text{opt}})$.

While it is theoretically reassuring that an optimal solution exists that is at least as good as all others and that this optimum can be characterized in dGL, such a solution is of limited practical usefulness since Eq. (3) cannot be executed without solving a game at runtime. Rather, we are interested in *explicit* solutions where I and G are quantifier-free real arithmetic formulas. There is no guarantee in general that such solutions exist that are also optimal, but our goal is to devise an algorithm to find them in the many cases where they exist or find safe approximations otherwise.

3.3 Controllable Invariants

The fact that S^{opt} is a solution can be characterized in logic with the notion of a controllable invariant that, at each of its points, admits some control action that keeps the plant in the invariant for one round. All lemmas and theorems throughout this paper are proved in the extended preprint [21, Appendix B].

Definition 2 (Controllable Invariant). *A controllable invariant is a formula I such that $\models I \rightarrow \text{safe}$ and $\models I \rightarrow \bigvee_i [\text{act}_i; \text{plant}] I$.*

From this perspective, I^{opt} can be seen as the largest controllable invariant.

Lemma 1. *I^{opt} is a controllable invariant and it is optimal in the sense that $\models I \rightarrow I^{\text{opt}}$ for any controllable invariant I .*

Moreover, not just I^{opt} , but *every* controllable invariant induces a solution. Indeed, given a controllable invariant I , we can define $\mathcal{G}(I) \equiv (i \mapsto [\text{act}_i; \text{plant}] I)$ for the control guards induced by I . $\mathcal{G}(I)$ chooses as the guard for each action act_i the modal condition ensuring that act_i preserves I after the plant.

Lemma 2. *If I is a controllable invariant, then $(I, \mathcal{G}(I))$ is a solution (Def. 1).*

Conversely, a controllable invariant can be derived from any solution.

Lemma 3. *If (I, G) is a solution, then $I' \equiv (\text{assum} \wedge I)$ is a controllable invariant. Moreover, we have $(I, G) \sqsubseteq (I', \mathcal{G}(I'))$.*

Solution comparisons w.r.t. \sqsubseteq reduce to implications for controllable invariants.

Lemma 4. *If I and I' are controllable invariants, then $(I, \mathcal{G}(I)) \sqsubseteq (I', \mathcal{G}(I'))$ if and only if $\models \text{assum} \rightarrow (I \rightarrow I')$.*

Taken together, these lemmas allow us to establish the optimality of S^{opt} .

Theorem 1. *S^{opt} is an optimal solution (i.e. a maximum w.r.t. \sqsubseteq) of Def. 1.*

This shows the roadmap for the rest of the paper: finding solutions to the control envelope synthesis problem reduces to finding controllable invariants that imply I^{opt} , which can be found by restricting the actions available to Demon in I^{opt} to guarantee safety, thereby *refining* the associated game.

3.4 One-Shot Fallback Refinement

The simplest refinement of I^{opt} is obtained when fixing a single fallback action to use in all states (if that is safe). A more general refinement considers different fallback actions in different states, but still only plays one such action forever.

Using the dGL axioms, any loop-free dGL formula whose ODEs admit solutions expressible in real arithmetic can be automatically reduced to an equivalent first-order arithmetic formula (in $\text{FOL}_{\mathbb{R}}$). An equivalent propositional arithmetic formula in $\mathcal{P}_{\mathbb{R}}$ can be computed via quantifier elimination (QE). For example:

$$\begin{aligned}
& [(v := 1 \cap v := -1); \{x' = v\}] x \neq 0 \\
& \equiv [v := 1 \cap v := -1] [\{x' = v\}] x \neq 0 && \text{by } [;] \\
& \equiv [v := 1] [\{x' = v\}] x \neq 0 \vee [v := -1] [\{x' = v\}] x \neq 0 && \text{by } [\cap] \\
& \equiv [\{x' = 1\}] x \neq 0 \vee [\{x' = -1\}] x \neq 0 && \text{by } [:=] \\
& \equiv (\forall t \geq 0 \ x + t \neq 0) \vee (\forall t \geq 0 \ x - t \neq 0) && \text{by } ['], [:=] \\
& \equiv x > 0 \vee x < 0 && \text{by QE} .
\end{aligned}$$

Even when a formula features nonsolvable ODEs, techniques exist to compute weakest preconditions for differential equations, with conservative approximations [38] or even exactly in some cases [35,8]. In the rest of this section and for most of this paper, we are therefore going to assume the existence of a **reduce** oracle that takes as an input a loop-free dGL formula and returns a quantifier-free arithmetic formula that is equivalent modulo some assumptions. Section 3.7 shows how to implement and optimize **reduce**.

Definition 3 (Reduction Oracle). *A reduction oracle is a function **reduce** that takes as an input a loop-free dGL formula F and an assumption $A \in \mathcal{P}_{\mathbb{R}}$. It returns a formula $R \in \mathcal{P}_{\mathbb{R}}$ along with a boolean flag **exact** such that the formula $A \rightarrow (R \rightarrow F)$ is valid, and if **exact** is true, then $A \rightarrow (R \leftrightarrow F)$ is valid as well.*

Back to our original problem, I^{opt} is not directly reducible since it involves a loop. However, conservative approximations can be computed by restricting the set of strategies that the Demon player is allowed to use. One extreme case allows Demon to only use a single action act_i repeatedly as a fallback (e.g. braking in the train example). In this case, we get a controllable invariant $[(\text{act}_i; \text{plant})^*] \text{ safe}$, which further simplifies into $[\text{act}_i; \text{plant}_{\infty}] \text{ safe}$ with

$$\text{plant}_{\infty} \equiv \{x' = f(x), t' = 1 \ \& \ \text{domain}\}$$

a variant of **plant** that never yields control. For this last step to be valid though, a technical assumption is needed on act_i , which we call *action permanence*.

Definition 4 (Action Permanence). *An action act_i is said to be permanent if and only if $(\text{act}_i; \text{plant}; \text{act}_i) \equiv (\text{act}_i; \text{plant})$, i.e., they are equivalent games.*

Intuitively, an action is *permanent* if executing it more than once in a row has no consequence for the system dynamics. This is true in the common case of actions that only assign constant values to control variables that are read but not modified by the plant, such as $a := A$ and $a := -B$ in Model 1.

Lemma 5. *If act_i is permanent, $\models [(\text{act}_i ; \text{plant})^*] \text{ safe} \leftrightarrow [\text{act}_i ; \text{plant}_\infty] \text{ safe}$.*

Our discussion so far identifies the following approximation to our original synthesis problem, where \mathbf{P} denotes the set of all indexes of permanent actions:

$$\begin{aligned} I^0 &\equiv [(\cap_{i \in \mathbf{P}} \text{act}_i) ; \text{plant}_\infty] \text{ safe}, \\ G_i^0 &\equiv [\text{act}_i ; \text{plant}] I^0. \end{aligned}$$

Here, I^0 encompasses all states from which the agent can guarantee safety indefinitely with a single permanent action. G^0 is constructed according to $\mathcal{G}(I^0)$ and only allows actions that are guaranteed to keep the agent within I^0 until the next control cycle. Note that I^0 degenerates to false in cases where there are no permanent actions, which does not make it less of a controllable invariant.

Theorem 2. *I^0 is a controllable invariant.*

Moreover, in many examples of interest, I^0 and I^{opt} are equivalent since an optimal fallback strategy exists that only involves executing a single action. This is the case in particular for Model 1, where

$$\begin{aligned} I^0 &\equiv [a := -B ; \{p' = v, v' = a \ \& \ v \geq 0\}] e - p > 0 \\ &\equiv e - p > v^2/2B \end{aligned}$$

characterizes all states at safe braking distance to the obstacle and G^0 associates the following guard to the acceleration action:

$$\begin{aligned} G_{a=A}^0 &\equiv [a := A ; \{p' = v, v' = a, t' = 1 \ \& \ v \geq 0 \wedge t \leq T\}] e - p > v^2/2B \\ &\equiv e - p > vT + AT^2/2 + (v + AT)^2/2B \end{aligned}$$

That is, accelerating is allowed if doing so is guaranteed to maintain sufficient braking distance until the next control opportunity. Section 3.6 discusses automatic generation of a proof that (I^0, G^0) is an optimal solution for Model 1.

3.5 Bounded Fallback Unrolling Refinement

In Section 3.4, we derived a solution by computing an underapproximation of I^{opt} where the fallback controller (played by Demon) is only allowed to use a one-shot strategy that picks a single action and plays it forever. Although this approximation is always safe and, in many cases of interest, happens to be exact, it does lead to a suboptimal solution in others. In this section, we allow the fallback controller to switch actions a bounded number of times before it plays one forever. There are still cases where doing so is suboptimal (imagine a car on a circular race track that is forced to maintain constant velocity). But this restriction is in line with the typical understanding of a fallback controller, whose mission is not to take over a system indefinitely but rather to maneuver it into a state where it can safely get to a full stop [32].

For all bounds $n \in \mathbb{N}$, we define a game where the fallback controller (played by Demon) takes at most n turns to reach the region I^0 in which safety is guaranteed indefinitely. During each turn, it picks a permanent action and chooses a time θ in advance for when it wishes to play its next move. Because the environment (played by Angel) has control over the duration of each control cycle, the fallback controller cannot expect to be woken up after time θ exactly. However, it can expect to be provided with an opportunity for its next move within the $[\theta, \theta + T]$ time window since the plant can never execute for time greater than T . Formally, we define I^n as follows:

$$\begin{aligned} I^n &\equiv [\text{step}^{\times n}; \text{forever}] \text{safe} & \text{forever} &\equiv (\cap_{i \in \mathbb{P}} \text{act}_i); \text{plant}_\infty \\ \text{step} &\equiv (\theta := *; ?\theta \geq 0)^d; (\cap_{i \in \mathbb{P}} \text{act}_i); \text{plant}_{\theta+T}; ?\text{safe}^d; ?t \geq \theta \end{aligned}$$

where $\text{plant}_{\theta+T}$ is the same as plant , except that the domain constraint $t \leq T$ is replaced by $t \leq \theta + T$. Equivalently, we can define I^n by induction as follows:

$$I^{n+1} \equiv I^n \vee [\text{step}] I^n \quad I^0 \equiv [\text{forever}] \text{safe}, \quad (5)$$

where the base case coincides with the definition of I^0 in Section 3.4. Importantly, I^n is a loop-free controllable invariant and so **reduce** can compute an explicit solution to the synthesis problem from I^n .

Theorem 3. *I^n is a controllable invariant for all $n \geq 0$.*

Theorem 3 establishes a nontrivial result since it overcomes the significant gap between the *fantasized* game that defines I^n and the *real* game being played by a time-triggered controller. The proof critically relies on the action permanence assumption along with a result [21, Lemma 6] establishing that ODEs preserve a specific form of reach-avoid property as a result of being deterministic.

Example. As an illustration, consider the example in Fig. 1 and Model 2 of a 2D robot moving in a corridor that forms an angle. The robot is only allowed to move left or down at a constant velocity and must not crash against a wall. Computing I^0 gives us the vertical section of the corridor, in which going down is a safe one-step fallback. Computing I^1 forces us to distinguish two cases. If the corridor is wider than the maximal distance travelled by the robot in a control cycle ($VT > 2R$), then the upper section of the corridor is controllable (with the exception of a dead-end that we prove to be uncontrollable in Section 3.6). On the other hand, if the corridor is too narrow, then I^1 is equivalent to I^0 . Formally, we have $I^1 \equiv (y > -R \wedge |x| < R) \vee (VT < 2R \wedge (x > -R \wedge |y| < R))$. Moreover, computing I^2 gives a result that is equivalent to I^1 . From this, we can conclude that I^1 is equivalent to I^n for all $n \geq 1$. Intuitively, it is optimal with respect to *any* finite fallback strategy (restricted to permanent actions).

The controllable invariant unrolling I^n has a natural stopping criterion.

Lemma 6. *If $I^n \leftrightarrow I^{n+1}$ is valid for some $n \geq 0$, then $I^n \leftrightarrow I^m$ is valid for all $m \geq n$ and $I^n \leftrightarrow I^\omega$ is valid where $I^\omega \equiv [\text{step}^\times; \text{forever}] \text{safe}$.*

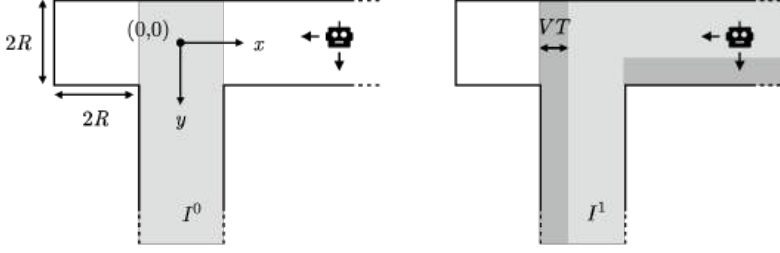


Fig. 1: Robot navigating a corridor (Model 2). A 2D robot must navigate safely within a corridor with a dead-end without crashing against a wall. The corridor extends infinitely on the bottom and on the right. The robot can choose between going left and going down with a constant speed V . The left diagram shows I^0 in gray. The right diagram shows I^1 under the additional assumption $VT < 2R$ (I^1 and I^0 are otherwise equivalent). A darker shade of gray is used for regions of I^1 where only one of the two available actions is safe according to G^1 .

Model 2 Robot navigating a corridor with framed solutions of holes.

assum	1	$V > 0 \wedge T > 0$
ctrlable	2	$\wedge \left[(y > -R \wedge x < R) \vee (VT < 2R \wedge (x > -R \wedge y < R)) \right] \rightarrow \{ \{$
ctrl	3	$(\text{?}[x > -R + VT]; v_x := -V; v_y := 0)$
	4	$\cup (\text{?}[y < R - VT \vee x < R]; v_x := 0; v_y := V) \} ;$
plant	5	$(t := 0; \{x' = v_x, y' = v_y, t' = 1 \ \& \ t \leq T\})$
safe	6	$\}^*]((x > -3R \wedge y < R) \vee (y > -R \wedge x < R))$

3.6 Proving Optimality via the Dual Game

Suppose one found a controllable invariant I using techniques from the previous section. To prove it optimal, one must show that $\models \text{assum} \rightarrow (I^{\text{opt}} \rightarrow I)$. By contraposition and $[\alpha] P \leftrightarrow \neg \langle \alpha \rangle \neg P$ ($[\cdot]$), this is equivalent to proving that:

$$\models \text{assum} \wedge \neg I \rightarrow \underbrace{\langle ((\cap_i \text{act}_i); \text{plant})^* \rangle}_{\neg I^{\text{opt}}} \neg \text{safe}. \quad (6)$$

We define the largest *uncontrollable* region $U^{\text{opt}} \equiv \neg I^{\text{opt}}$ as the right-hand side of implication 6 above. Intuitively, U^{opt} characterizes the set of all states from which the environment (played by Angel) has a winning strategy against the controller (played by Demon) for reaching an unsafe state. In order to prove the optimality of I , we compute a sequence of increasingly strong approximations U of U^{opt} such that $U \rightarrow U^{\text{opt}}$ is valid. We do so via an iterative process, in the spirit of how we approximate I^{opt} via bounded fallback unrolling (Section 3.5), although the process can be guided by the knowledge of I this time. If at any point we manage to prove that $\text{assum} \rightarrow (I \vee U)$ is valid, then I is optimal.

One natural way to compute increasingly good approximations of U^{opt} is via loop unrolling. The idea is to improve approximation U by adding states from where the environment can reach U by running the control loop once, formally, $\langle (\cap_i \text{act}_i); \text{plant} \rangle U$. This unrolling principle can be useful. However, it only augments U with new states that can reach U in time T at most. So it cannot alone prove optimality in cases where violating safety from an unsafe state takes an unbounded amount of time.

For concreteness, let us prove the optimality of I^0 in the case of Model 1. In [34] essentially the following statement is proved when arguing for optimality: $\models \text{assum} \wedge \neg I^0 \rightarrow \langle (a := -B; \text{plant})^* \rangle \neg \text{safe}$. This is identical to our optimality criterion from Eq. (6), except that Demon’s actions are restricted to braking. Intuitively, this restriction is sound since accelerating always makes things worse as far as safety is concerned. If the train cannot be saved with braking alone, adding the option to accelerate will not help a bit. In this work, we propose a method for formalizing such arguments within dGL to arbitrary systems.

Our idea for doing so is to consider a system made of two separate copies of our model. One copy has all actions available whereas the other is only allowed a single action (e.g. braking). Given a safety metric m (i.e. a term m such that $\models m \leq 0 \rightarrow \neg \text{safe}$), we can then formalize the idea that “action i is always better w.r.t safety metric m ” within this joint system.

Definition 5 (Uniform Action Optimality). *Consider a finite number of discrete dL programs α_i and $p \equiv \{x' = f(x) \ \& \ Q\}$. Let $V = \text{BV}(p) \cup \bigcup_i \text{BV}(\alpha_i)$ be the set of all variables written by p or some α_i . For any term θ and integer n , write $\theta^{(n)}$ for the term that results from θ by renaming all variables $v \in V$ to a fresh tagged version $x^{(n)}$. Using a similar notation for programs and formulas, define $p^{(1,2)} \equiv \{(x^{(1)})' = f(x^{(1)}), (x^{(2)})' = f(x^{(2)}) \ \& \ Q^{(1)} \wedge Q^{(2)}\}$. We say that action j is uniformly optimal with respect to safety metric m if and only if:*

$$\models m^{(1)} \geq m^{(2)} \rightarrow [\alpha_j^{(1)}; (\cup_i \alpha_i^{(2)})]; p^{(1,2)}] m^{(1)} \geq m^{(2)}.$$

$\text{best}_j((\alpha_i)_i, p, m)$ denotes that action j is uniformly optimal with respect to m for actions α_i and dynamics p .

With such a concept in hand, we can formally establish the fact that criterion Eq. (6) can be relaxed in the existence of uniformly optimal actions.

Theorem 4. *Consider a finite number of discrete dL programs α_i such that $\models \langle \alpha_i \rangle \text{true}$ for all i and $p \equiv \{x' = f(x) \ \& \ q \geq 0\}$. Then, provided that $\text{best}_j((\alpha_i)_i, p, m)$ and $\text{best}_j((\alpha_i)_i, p, -q)$ (no other action stops earlier because of the domain constraint), we have:*

$$\models \langle ((\cap \alpha_i); p)^* \rangle m \leq 0 \leftrightarrow \langle (\alpha_j; p)^* \rangle m \leq 0 \ .$$

A general heuristic for leveraging Theorem 4 to grow U automatically works as follows. First, it considers $R \equiv \text{assum} \wedge \neg I \wedge \neg U$ that characterizes states that are not known to be controllable or uncontrollable. Then, it picks a disjunct $\bigwedge_j R_j$ of

the disjunctive normal form of R and computes a forward invariant region V that intersects with it: $V \equiv \bigwedge_j \{R_j : \text{assum}, R_j \vdash [(\cup_i \text{act}_i); \text{plant}] R_j\}$. Using V as an assumption to simplify $\neg U$ may suggest metrics to be used with Theorem 4. For example, observing $\models V \rightarrow (\neg U \rightarrow (\theta_1 > 0 \wedge \theta_2 > 0))$ suggests picking metric $m \equiv \min(\theta_1, \theta_2)$ and testing whether $\text{best}_j(\text{act}, p, m)$ is true for some action j . If such a uniformly optimal action exists, then U can be updated as $U \leftarrow U \vee (V \wedge \langle (\text{act}_j; \text{plant})^* \rangle m \leq 0)$. The solution I^1 for the corridor (Model 2) can be proved optimal automatically using this heuristic in combination with loop unrolling.

3.7 Implementing the Reduction Oracle

The CESAR algorithm assumes the existence of a *reduction oracle* that takes as an input a loop-free dGL formula and attempts to compute an equivalent formula within the fragment of propositional arithmetic. When an exact solution cannot be found, an implicant is returned instead and flagged appropriately (Def. 3). This section discusses our implementation of such an oracle.

As discussed in Section 3.4, exact solutions can be computed systematically when all ODEs are solvable by first using the dGL axioms to eliminate modalities and then passing the result to a *quantifier elimination algorithm* for first-order arithmetic [9,42]. Although straightforward in theory, a naïve implementation of this idea hits two practical barriers. First, quantifier elimination is expensive and its cost increases rapidly with formula complexity [11,44]. Second, the output of existing QE implementations can be unnecessarily large and redundant. In iterated calls to the reduction oracle, these problems can compound each other.

To alleviate this issue, our implementation performs *eager simplification* at intermediate stages of computation, between some axiom application and quantifier-elimination steps. This optimization significantly reduces output solution size and allows CESAR to solve a benchmark that would otherwise timeout after 20 minutes in 26s. [21, Appendix E] further discusses the impact of eager simplification. Still, the doubly exponential complexity of quantifier elimination puts a limit on the complexity of problems that CESAR can currently tackle.

In the general case, when ODEs are not solvable, our reduction oracle is still often able to produce *approximate* solutions using differential invariants generated automatically by existing tools [38]. Differential invariants are formulas that stay true throughout the evolution of an ODE system.⁴ To see how they apply, consider the case of computing $\text{reduce}(\{x' = f(x)\} P, A)$ where P is the post-condition formula that must be true after executing the differential equation, and A is the assumptions holding true initially. Suppose that formula $D(x)$ is a differential invariant such that $D(x) \rightarrow P$ is valid. Then, a precondition sufficient to ensure that P holds after evolution is $A \rightarrow D(x)$. For example, to compute the precondition for the dynamics of the **parachute** benchmark, our reduction oracle first uses the Pegasus tool [38] to identify a Darboux polynomial, suggesting

⁴ dGL provides ways to reason about differential invariants without solving the corresponding differential equation. For example, for an invariant of the form $e = 0$, the differential invariant axiom is $\{x' = f(x)\} e = 0 \leftrightarrow (e = 0 \wedge \{x' = f(x)\} e' = 0)$.

an initial differential invariant D_0 . Once we have D_0 , the additional information required to conclude post condition P is $D_0 \rightarrow P$. To get an invariant formula that implies $D_0 \rightarrow P$, eliminate all the changing variables $\{x, v\}$ in the formula $\forall x \forall v (D_0 \rightarrow P)$, resulting in a formula D_1 . D_1 is a differential invariant since it features no variable that is updated by the ODEs. Our reduction oracle returns $D_0 \wedge D_1$, an invariant that entails postcondition P .

3.8 The CESAR Algorithm

The CESAR algorithm for synthesizing control envelopes is summarized in Algorithm 1. It is expressed as a generator that yields a sequence of solutions with associated optimality guarantees. Possible guarantees include “*sound*” (no optimality guarantee, only soundness), “*k-optimal*” (sound and optimal w.r.t all k -switching fallbacks with permanent actions), “ *ω -optimal*” (sound and optimal w.r.t all finite fallbacks with permanent actions) and “*optimal*” (sound and equivalent to S^{opt}). Line 11 performs the optimality test described in Section 3.6. Finally, Line 10 performs an important soundness check for the cases where an approximation has been made along the way of computing (I^n, G^n) . In such cases, I is not guaranteed to be a controllable invariant and thus Case (2) of Def. 1 must be checked explicitly.

When given a problem with solvable ODEs and provided with a complete QE implementation within **reduce**, CESAR is guaranteed to generate a solution in finite time with an “*n-optimal*” guarantee at least (n being the unrolling limit).

4 Benchmarks and Evaluation

To evaluate our approach to the Control Envelope Synthesis problem, we curate a benchmark suite with diverse optimal control strategies. As Table 2 summarizes, some benchmarks have non-solvable dynamics, while others require a sequence of clever control actions to reach an optimal solution. Some have *state-dependent fallbacks* where the current state of the system determines which action is “safer”, and some are drawn from the literature. We highlight a couple of benchmarks here. See [21, Appendix D] for a discussion of the full suite and the synthesized results, and [20] for the benchmark files and evaluation scripts.

Power Station is an example where the optimal control strategy involves two switches, corresponding to two steps of unrolling. A power station can either produce power or dispense it to meet a quota, but never give out more than it has produced. Charging is the fallback action that is safe for all time *after* the station has dispensed enough power. However, to cover all controllable states, we need to switch at least two times, so that the power station has a chance to produce energy and then dispense it, before settling back on the safe fallback. **Parachute** is an example of a benchmark with non-solvable, hyperbolic dynamics. A person jumps off a plane and can make an irreversible choice to open their parachute. The objective is to stay within a maximum speed that is greater than the terminal velocity when the parachute is open.

Algorithm 1 CESAR: Control Envelope Synthesis via Angelic Refinements

```

1: Input: a synthesis problem (as defined in Section 3.1), an unrolling limit  $n$ .
2: Remark:  $\text{valid}$  is defined as  $\text{valid}(F, A) \equiv (\text{first}(\text{reduce}(\neg F, A)) = \text{false})$ .
3:  $k \leftarrow 0$ 
4:  $I, e_I \leftarrow \text{reduce}([\text{forever}] \text{ safe}, \text{ assum})$ 
5: while  $k \leq n$  do
6:    $e_G \leftarrow \text{true}$ 
7:   for each  $i$  do
8:      $G_i, e \leftarrow \text{reduce}([\text{act}_i; \text{plant}] I, \text{ assum})$ 
9:      $e_G \leftarrow e_G$  and  $e$ 
10:  if  $(e_G$  and  $e_I)$  or  $\text{valid}(I \rightarrow \bigvee_i G_i, \text{ assum})$  then
11:    if  $e_G$  and  $\text{optimal}(I)$  then
12:      yield  $((I, G), \text{"optimal"})$ 
13:    return
14:    else if  $e_G$  and  $e_I$  then yield  $((I, G), \text{"k-optimal"})$ 
15:    else yield  $((I, G), \text{"sound"})$ 
16:   $I', e \leftarrow \text{reduce}(I \vee [\text{step}] I, \text{ assum})$ 
17:   $e_I \leftarrow e_I$  and  $e$ 
18:  if  $e_G$  and  $e_I$  and  $\text{valid}(I' \rightarrow I, \text{ assum})$  then
19:    yield  $((I, G), \text{"\omega-optimal"})$ 
20:  return
21:   $I \leftarrow I'$ 
22:   $k \leftarrow k + 1$ 

```

We implement CESAR in Scala, using Mathematica for simplification and quantifier elimination, and evaluate it on the benchmarks. Simplification is an art [25,23]. We implement additional simplifiers with the Egg library [45] and SMT solver z3 [30]. Experiments were run on a 32GB RAM M2 MacBook Pro machine. CESAR execution times average over 5 runs.

CESAR synthesis is automatic. The optimality tests were computed manually. Table 2 summarizes the result of running CESAR. Despite a variety of different control challenges, CESAR is able to synthesize safe and in some cases also optimal safe control envelopes within a few minutes. As an extra step of validation, synthesized solutions are checked by the hybrid system theorem prover KeYmaera X [16]. All solutions are proved correct, with verification time as reported in the last column of Table 2.

5 Related Work

Hybrid controller synthesis has received significant attention [26,41,7], with popular approaches using temporal logic [5,7,46], games [31,43], and CEGIS-like guidance from counterexamples [39,1,37,10]. CESAR, however, solves the different problem of synthesizing control *envelopes* that strive to represent not one but *all* safe controllers of a system. Generating *valid* solutions is not an issue (a trivial solution always exists that has an empty controllable set). The real challenge is *optimality* which imposes a higher order constraint because it reasons

Table 2: Summary of CESAR experimental results

Benchmark	Synthesis Time (s)	Checking Time (s)	Optimal	Needs Unrolling	Non Solvable Dynamics
ETCS Train [34]	14	9	✓		
Sled	20	8	✓		
Intersection	49	44	✓		
Parachute [15]	46	8			✓
Curvebot	26	9			✓
Coolant	49	20	✓	✓	
Corridor	20	8	✓	✓	
Power Station	26	17	✓	✓	

about the relationship between possible valid solutions, and cannot, e.g., fit in the CEGIS quantifier alternation pattern $\exists\forall$. So simply adapting existing controller synthesis techniques does not solve symbolic control envelope synthesis.

Safety shields computed by numerical methods [2,13,24] serve a similar function to our *control envelopes* and can handle dynamical systems that are hard to analyze symbolically. However, they scale poorly with dimensionality and do not provide rigorous formal guarantees due to the need of discretizing continuous systems. Compared to our symbolic approach, they cannot handle unbounded state spaces (e.g. our infinite corridor) nor produce shields that are parametric in the model’s parameters without hopelessly increasing dimensionality.

On the optimality side, a systematic but manual process was used to design a safe European Train Control System (ETCS) and justify it as optimal with respect to specific train criteria [34]. Our work provides the formal argument filling the gap between such case-specific criteria and end-to-end optimality. CESAR is more general and automatic.

6 Conclusion

This paper presents the CESAR algorithm for Control Envelope Synthesis via Angelic Refinements. It is the first approach to automatically synthesize symbolic control envelopes for hybrid systems. The synthesis problem and optimal solution are characterized in differential game logic. Through successive refinements, the optimal solution in game logic is translated into a controllable invariant and control conditions. The translation preserves safety. For the many cases where refinement additionally preserves optimality, an algorithm to test optimality of the result post translation is presented. The synthesis experiments on a benchmark suite of diverse control problems demonstrate CESAR’s versatility. For future work, we plan to extend to additional control shapes, and to exploit the synthesized safe control envelopes for reinforcement learning.

References

1. Abate, A., Bessa, I., Cordeiro, L.C., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica* **57**(1-2), 223–244 (2020). doi: 10.1007/s00236-019-00359-1
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. *Proceedings of the Aaai Conference on Artificial Intelligence* **32** (2018). doi: 10.1609/aaai.v32i1.11797
3. Alur, R.: *Principles of Cyber-Physical Systems*. MIT Press, Cambridge (2015)
4. Ames, A.D., Coogan, S., Egerstedt, M., Notomista, G., Sreenath, K., Tabuada, P.: Control barrier functions: Theory and applications. In: 17th European Control Conference, ECC 2019, Naples, Italy, June 25–28, 2019. pp. 3420–3431. IEEE (2019). doi: 10.23919/ECC.2019.8796030
5. Antoniotto, M., Mishra, B.: Discrete event models+temporal logic=supervisory controller: automatic synthesis of locomotion controllers. In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. vol. 2, pp. 1441–1446 vol.2 (1995). doi: 10.1109/ROBOT.1995.525480
6. Basile, G., Marro, G.: Controlled and conditioned invariant subspaces in linear system theory. *Journal of Optimization Theory and Applications* **3**, 306–315 (05 1969). doi: 10.1007/BF00931370
7. Belta, C., Yordanov, B., Gol, E.A.: *Formal Methods for Discrete-Time Dynamical Systems*. Springer Cham (2017)
8. Boreale, M.: Complete algorithms for algebraic strongest postconditions and weakest preconditions in polynomial ODE's. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) *SOFSEM 2018: Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science*, Krems, Austria, January 29 - February 2, 2018, *Proceedings. LNCS*, vol. 10706, pp. 442–455. Springer (2018)
9. Caviness, B.F., Johnson, J.R.: *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media (2012)
10. Dai, H., Landry, B., Pavone, M., Tedrake, R.: Counter-example guided synthesis of neural network lyapunov functions for piecewise linear systems. 2020 59th IEEE Conference on Decision and Control (CDC) pp. 1274–1281 (2020)
11. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. *J. Symb. Comput.* **5**(1/2), 29–35 (1988)
12. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) *Handbook of Model Checking*, pp. 1047–1110. Springer, Cham (2018). doi: 10.1007/978-3-319-10575-8_30
13. Fisac, J., Akametalu, A., Zeilinger, M., Kaynama, S., Gillula, J., Tomlin, C.: A general safety framework for learning-based control in uncertain robotic systems. *Ieee Transactions on Automatic Control* **64**, 2737–2752 (2019). doi: 10.1109/tac.2018.2876389
14. Freiburger, F., Schupp, S., Hermanns, H., Ábrahám, E.: Controller verification meets controller code: A case study. In: *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*. p. 98–103. MEMOCODE '21, Association for Computing Machinery, New York, NY, USA (2021). doi: 10.1145/3487212.3487337

15. Fulton, N., Mitsch, S., Bohrer, R., Platzer, A.: Bellerophon: Tactical theorem proving for hybrid systems. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP. LNCS, vol. 10499, pp. 207–224. Springer (2017). doi: [10.1007/978-3-319-66107-0_14](https://doi.org/10.1007/978-3-319-66107-0_14)
16. Fulton, N., Mitsch, S., Quesel, J.D., Völz, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: CADE. pp. 527–538 (2015). doi: [10.1007/978-3-319-21401-6_36](https://doi.org/10.1007/978-3-319-21401-6_36)
17. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence. AAAI’18/IAAI’18/EAAI’18, AAAI Press (2018)
18. Ghosh, B.K.: Controlled invariant and feedback controlled invariant subspaces in the design of a generalized dynamical system. In: 1985 24th IEEE Conference on Decision and Control. pp. 872–873 (1985). doi: [10.1109/CDC.1985.268620](https://doi.org/10.1109/CDC.1985.268620)
19. Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Case study: Verifying the safety of an autonomous racing car with a neural network controller. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control. HSCC ’20, Association for Computing Machinery, New York, NY, USA (2020). doi: [10.1145/3365365.3382216](https://doi.org/10.1145/3365365.3382216)
20. Kabra, A., Laurent, J., Mitsch, S., Platzer, A.: Control Envelope Synthesis via Angelic Refinements (CESAR): Artifact (1 2024). doi: [10.6084/m9.figshare.24922896.v1](https://doi.org/10.6084/m9.figshare.24922896.v1), https://figshare.com/articles/software/Control_Envelope_Synthesis_via_Angelic_Refinements_CESAR_Artifact/24922896
21. Kabra, A., Laurent, J., Mitsch, S., Platzer, A.: Cesar: Control envelope synthesis via angelic refinements (2023). doi: <https://doi.org/10.48550/arXiv.2311.02833>, arXiv:2311.02833
22. Kabra, A., Mitsch, S., Platzer, A.: Verified train controllers for the federal railroad administration train kinematics model: Balancing competing brake and track forces. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 4409–4420 (2022). doi: [10.1109/TCAD.2022.3197690](https://doi.org/10.1109/TCAD.2022.3197690)
23. Knuth, D.E.: The Art of Computer Programming. Addison Wesley Longman Publishing Co., Inc., USA (1997)
24. Kochenderfer, M.J., Holland, J.E., Chryssanthacopoulos, J.P.: Next generation airborne collision avoidance system. Lincoln Laboratory Journal **19**(1), 17–33 (2012)
25. Lara, M., López, R., Pérez, I., San-Juan, J.F.: Exploring the long-term dynamics of perturbed keplerian motion in high degree potential fields. Communications in Nonlinear Science and Numerical Simulation **82**, 105053 (2020). doi: <https://doi.org/10.1016/j.cnsns.2019.105053>, <https://www.sciencedirect.com/science/article/pii/S1007570419303727>
26. Liu, S., Trivedi, A., Yin, X., Zamani, M.: Secure-by-construction synthesis of cyber-physical systems. Annual Reviews in Control **53**, 30–50 (2022). doi: <https://doi.org/10.1016/j.arcontrol.2022.03.004>
27. Lunze, J., Lamnabhi-Lagarrigue, F. (eds.): Handbook of Hybrid Systems Control: Theory, Tools, Applications. Cambridge Univ. Press, Cambridge (2009). doi: [10.1017/CB09780511807930](https://doi.org/10.1017/CB09780511807930)
28. Mitra, S.: Verifying Cyber-Physical Systems: A Path to Safe Autonomy. MIT Press (2021)
29. Mitsch, S., Platzer, A.: Modelplex: verified runtime validation of verified cyber-physical system models. Formal Methods Syst. Des. **49**(1-2), 33–74 (2016). doi: [10.1007/s10703-016-0241-z](https://doi.org/10.1007/s10703-016-0241-z)

30. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
31. Nerode, A., Yakhnis, A.: Modelling hybrid systems as games. In: *Decision and Control, 1992., Proceedings of the 31st IEEE Conference on*. pp. 2947–2952 vol.3 (1992). doi: 10.1109/CDC.1992.371272
32. Pek, C., Althoff, M.: Fail-safe motion planning for online verification of autonomous vehicles using convex optimization. *IEEE Transactions on Robotics* **37**(3), 798–814 (2020)
33. Platzer, A.: *Logical Foundations of Cyber-Physical Systems*. Springer, Cham (2018). doi: 10.1007/978-3-319-63588-0
34. Platzer, A., Quesel, J.: European train control system: A case study in formal verification. In: *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*. pp. 246–265 (2009). doi: 10.1007/978-3-642-10373-5_13
35. Platzer, A., Tan, Y.K.: Differential equation invariance axiomatization. *Journal of the ACM (JACM)* **67**(1), 1–66 (2020)
36. Qian, M., Mitsch, S.: Reward shaping from hybrid systems models in reinforcement learning. In: Rozier, K.Y., Chaudhuri, S. (eds.) *NFM. LNCS*, vol. 13903. Springer (2023)
37. Ravanbakhsh, H., Sankaranarayanan, S.: Robust controller synthesis of switched systems using counterexample guided framework. In: *2016 International Conference on Embedded Software, EMSOFT 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. pp. 8:1–8:10 (2016). doi: 10.1145/2968478.2968485
38. Sogokon, A., Mitsch, S., Tan, Y.K., Cordwell, K., Platzer, A.: Pegasus: Sound continuous invariant generation. *Form. Methods Syst. Des.* **58**(1), 5–41 (2022). doi: 10.1007/s10703-020-00355-z, special issue for selected papers from FM’19
39. Solar-Lezama, A.: Program sketching. *STTT* **15**(5-6), 475–495 (2013). doi: 10.1007/s10009-012-0249-7
40. Squires, E., Pierpaoli, P., Egerstedt, M.: Constructive barrier certificates with applications to fixed-wing aircraft collision avoidance. In: *2018 IEEE Conference on Control Technology and Applications (CCTA)*. pp. 1656–1661 (2018). doi: 10.1109/CCTA.2018.8511342
41. Tabuada, P.: *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, Berlin (2009). doi: 10.1007/978-1-4419-0224-5
42. Tarski, A.: A decision method for elementary algebra and geometry. In: Caviness, B.F., Johnson, J.R. (eds.) *Quantifier Elimination and Cylindrical Algebraic Decomposition*. pp. 24–84. Springer Vienna, Vienna (1998)
43. Tomlin, C.J., Lygeros, J., Sastry, S.: A game theoretic approach to controller design for hybrid systems. *Proc. IEEE* **88**(7), 949–970 (2000). doi: 10.1109/5.871303
44. Weispfenning, V.: The complexity of linear problems in fields. *J. Symb. Comput.* **5**(1-2), 3–27 (1988)
45. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* **5**(POPL) (jan 2021). doi: 10.1145/3434304, <https://doi.org/10.1145/3434304>
46. Yang, S., Yin, X., Li, S., Zamani, M.: Secure-by-construction optimal path planning for linear temporal logic tasks. In: *2020 59th IEEE Conference on Decision and Control (CDC)*. pp. 4460–4466 (2020). doi: 10.1109/CDC42340.2020.9304153

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 Decidability



Answering Temporal Conjunctive Queries over Description Logic Ontologies for Situation Recognition in Complex Operational Domains^{*}

Lukas Westhofen¹(✉) , Christian Neurohr¹ , Jean Christoph Jung² ,
and Danielneider^{2,3}

¹ German Aerospace Center (DLR) e.V., Institute of Systems Engineering for Future Mobility, Oldenburg, Germany

lukas.westhofen@dlr.de, christian.neurohr@dlr.de

² TU Dortmund University, Dortmund, Germany

jean.jung@tu-dortmund.de, daniel.neider@tu-dortmund.de

³ Center for Trustworthy Data Science and Security, University Alliance Ruhr, Dortmund, Germany

Abstract. For developing safe automated systems, recognizing safety-critical situations in data from their complex operational domain is imperative. This capability is, for example, essential when evaluating the system’s conformance to specified requirements in test run data. The requirements involve a temporal dimension, as the system operates over time. Moreover, the generated data are usually relational and require additional background knowledge about the domain for correctly recognizing the situation. This fact makes propositional temporal logics, an established tool, unsuitable for the task. We address this issue by developing a tailored temporal logic to query for situations in relational data over complex domains. Our language combines mission-time linear temporal logic with conjunctive queries to access time-stamped data with background knowledge formulated in an expressive description logic. Currently, however, no tools exist for answering queries in such settings. We hence also contribute an implementation in the logic reasoner OPENLLET, leveraging the efficacy of well-established conjunctive query answering. Moreover, we present a benchmark generator in the setting of automated driving and demonstrate that our tool performs well when tasked with recognizing safety-critical situations in road traffic.

Keywords: Temporal Conjunctive Queries · Description Logics · Temporal Logics.

^{*} This work was partially funded by the German Federal Ministries of Education and Research (‘AutoDevSafeOps’) and Economic Affairs and Climate Action (‘VVM – Verification & Validation Methods for Automated Vehicles Level 4 and 5’).

1 Introduction

Recent technological advances in, e.g., sensors and computer vision, gave updraft to the development of automated systems performing safety-critical tasks in complex domains. These systems are expected to safely operate without human intervention in these contexts. Consider, for example, automated driving systems (ADSs), where the responsibility of safely navigating the environment lies fully with the system [35]. The combination of their safety-critical nature and the complex operational domain makes it hard to guarantee the absence of unreasonable risk before public release, which is, however, required by many homologation authorities. Alas, correct-by-design techniques are rendered inapplicable by the high system complexity. Thus, manufactures must resort to empirically assessing the system's risk prior to deployment. As automated systems interact with their environment over time, a promising approach for risk assessment is to decompose the complex operational domain into finite-time sequences ('scenarios') [34]. Safety requirements – aiming to mitigate unreasonable risks – are then specified for these scenarios. Hence, a formal specification of the actors' temporal behavior becomes essential. An exemplary requirement reads as follows: 'In situations where the absence of pedestrians is not guaranteed, adapt the speed appropriately.' Note that this rule consists of a premise (the situation) and a consequence (the behavior). The number of situations to write requirements for can be enormous, e.g., *occlusions* [42], violating the *safety distance* [43], and maneuvers such as *passing parking vehicles* [11]. Due to their large number, testing the most widely used option for verification, i.e., to check the system's conformance with requirements. For this, data of test runs of the system operating within its environment are recorded. Adherence to the requirements is then evaluated by recognizing the situation ('no guaranteed absence of pedestrians') and testing for the implied behavior ('adapted speed'). We argue that this approach has three requirements:

Relational and Temporal Domain Formally modelling traffic situations inherently requires a relational language since they refer to individuals and their relationships, e.g., *drives*. Moreover, the number of individuals is not fixed beforehand. Finally, scenarios over such situations involve the description of temporal aspects. A typical example is the process of overtaking.

Rich Background Knowledge We do not assume that the data is complete in the sense that we can observe all facts about all individuals. Instead, we assume to have rich knowledge about the relations used in the situation descriptions. Examples for this are:

- a **Driver** is equivalent to a **Human** which *drives* some **Vehicle**, or
- a **Driver** is never a **Pedestrian**.

Such knowledge must be included since otherwise situations may not be correctly recognized in the data and test evaluation produces false results.

Formal Specifications of Properties It is established that specifying and testing requirements benefits greatly from formal approaches. Standard requirement formalization languages, like linear temporal logic, are however propositional and thus unsuitable for our purposes.

An established way to address the first two aspects is to model situations via *temporal knowledge bases* $\mathcal{K} = (\mathcal{O}, \mathcal{D})$ which consist of a domain ontology \mathcal{O} that describes the background knowledge and a temporal database \mathcal{D} that describes the evolution of the situation over time. Formally, \mathcal{D} is a sequence $\mathcal{D} = (\mathcal{D}_0, \dots, \mathcal{D}_n)$ of time-stamped databases. Note that, in using temporal knowledge bases, we adopt the *open world assumption* (OWA), which intuitively says that the true facts are not only those in \mathcal{D} but those that are *entailed* by \mathcal{O} and \mathcal{D} .

As to address the third aspect above, i.e., to formally specify properties, we use a suitable extension of linear temporal logic (LTL). Recall that LTL is a language for describing properties over a set of propositions by using modalities such as $\Diamond\varphi$ (φ holds *eventually*), $\Box\varphi$ (φ holds *globally*), $\varphi_1 \mathcal{U} \varphi_2$ (φ_1 holds *until* φ_2), and $\bigcirc\varphi$ (φ holds in the *next* step). Unfortunately, this does not suffice when working over relational data. A natural option to extend LTL in the required way is to replace propositions by *queries*. In this work, we use conjunctive queries (CQs). CQs are one of the most common query language for databases and expressively equivalent to the SELECT-FROM-WHERE fragment of SQL. For example, we can ask for all drivers d of a vehicle by the CQ $\exists v. \text{Vehicle}(v) \wedge \text{drives}(d, v)$ with one existentially quantified variable v and one answer variable d . In terms of the temporal expressivity, our application further requires that

- (1) we operate on finite traces whose length is bounded by the length of the temporal database \mathcal{D} specified in the temporal knowledge base,
- (2) as duration constraints are used in specifications, e.g., to distinguish maneuvers of certain lengths, we incorporate metric operators, and
- (3) we analyze the data a-posteriori. Hence we are not in a run-time verification setting and require only future time operators.

We term the resulting language *metric temporal conjunctive queries* (MTCQs), which features both unbounded and bounded future time operators over finite traces and uses CQs in its atoms and is based on Mission-Time LTL (MLTL) [29]. MTCQs can, for example, express properties like $\Phi_0^{ex}(x) = \Diamond \neg \text{Pedestrian}(x)$, asking for all individuals x that are eventually not a pedestrian. A more involved MTCQ asking for all x that move past a parking vehicle y on a two-lane road is

$$\begin{aligned} \Phi_1^{ex}(x, y) = & \Box(\exists r. \text{Vehicle}(x) \wedge 2_Lane_Road(r) \wedge \text{intersects}(r, x) \wedge \\ & \text{Parking_Vehicle}(y)) \wedge \Diamond(\text{in_front_of}(y, x) \wedge \bigcirc \\ & ((\text{in_proximity}(x, y) \wedge \text{to_the_side_of}(y, x)) \mathcal{U} \text{behind}(y, x))). \end{aligned}$$

Recognizing such a situation for checking a requirement translates to the task of evaluating an MTCQ $\Phi(\vec{x})$ with answer variables \vec{x} over a temporal knowledge base \mathcal{K} . Informally, if we want to verify that a tuple of individuals \vec{a} conforms to some specification $\Phi(\vec{x})$ in a situation $\mathcal{K} = (\mathcal{O}, \mathcal{D})$, we have to check whether the entailment $(\mathcal{O}, \mathcal{D}) \models \Phi(\vec{a})$ is true, cf. Section 3 for precise definitions.

This task obviously depends on the chosen ontology language. For this, we use description logics (DLs), an established knowledge representation formalism, which offers a good compromise between complexity and expressivity [10]. Our

approach works up to the $\mathcal{SRIQ}^{(D)}$ fragment of DLs. It is close to the formalism behind the Web Ontology Language (OWL) 2, an expressive and widespread DL language. The mentioned task of entailment has been studied for DL temporal knowledge bases and a different yet related extension of LTL [8], cf. Section 2.

We now illustrate this setup by means of a simple example. A DL ontology \mathcal{O} is a set of concept inclusions $\mathbf{C} \sqsubseteq \mathbf{D}$ for concept descriptions \mathbf{C} and \mathbf{D} . We also write $\mathbf{C} \equiv \mathbf{D}$ to denote concept equivalence. DLs allow for arbitrary names as basic concepts. We have special names for nothing (\perp) and all things (\top). Besides concepts, DLs also allow so-called roles (relations) between concepts. From these, we can inductively build new concepts. For an example ontology \mathcal{O}^{ex} , we might state that every driver is a human by $\mathbf{Driver} \sqsubseteq \mathbf{Human} \in \mathcal{O}^{ex}$. As to illustrate the combination of roles and concepts we define drivers as the intersection (using the \sqcap -operator) of all humans and all things that drive some (using the \exists -operator) vehicle, written as $\mathbf{Driver} \equiv \mathbf{Human} \sqcap \exists \mathbf{drives.Vehicle} \in \mathcal{O}^{ex}$. We can use \perp to express that pedestrians and drivers are disjoint: $\mathbf{Driver} \sqcap \mathbf{Pedestrian} \sqsubseteq \perp \in \mathcal{O}^{ex}$.

These operators may be enough for simple domains. However, knowledge about relations in complex domains is often involved, in which case even more expressive operators can be allowed. For example, the MTCQ Φ_1^{ex} requires recognizing situations of passing parking vehicles. Here, expressive DLs allow modeling two-lane roads to have exactly two lanes (by the concept $\mathbf{=2has_lane.Lane}$) and be a road (by the concept $\mathbf{Road} \sqcap \mathbf{=2has_lane.Lane}$). Moreover, parking vehicles are standing (with a speed of the datatype literal 0.0) dynamical objects on a parking spot. This is expressed by the following DL ontology:

- $\mathbf{2_Lane_Road} \equiv \mathbf{Road} \sqcap \mathbf{=2has_lane.Lane}$
- $\mathbf{Vehicle} \sqcap \mathbf{Standing_Dynamical_Object} \sqcap \exists \mathbf{intersects.Parking_Spot} \sqsubseteq \mathbf{Parking_Vehicle}$
- $\mathbf{Parking_Spot} \equiv \mathbf{Parking_Lane} \sqcup \mathbf{Walkway}$
- $\mathbf{Standing_Dynamical_Object} \equiv \mathbf{Dynamical_Object} \sqcap \exists \mathbf{has_speed.\{0.0\}}$

Let us now use the simple example to give an intuition on the semantics of MTCQs over DL ontologies. First, we create an exemplary database with facts over so-called individuals (concrete objects that are perceived). For example, we can assert for the first time point that the individual \mathbf{h} is a human driving the individual \mathbf{v} , a vehicle, by writing the facts as $\mathcal{D}_0^{ex} = \{\mathbf{Human}(\mathbf{h}), \mathbf{drives}(\mathbf{h}, \mathbf{v}), \mathbf{Vehicle}(\mathbf{v})\}$. Next, we may perceive $\mathcal{D}_1^{ex} = \emptyset$, i.e., no information at all. Together with the ontology, it forms a temporal knowledge base $\mathcal{K}^{ex} = (\mathcal{O}^{ex}, (\mathcal{D}_0^{ex}, \mathcal{D}_1^{ex}))$. If we query \mathcal{K}^{ex} w.r.t. $\Phi_0^{ex}(x) = \Diamond \neg \mathbf{Pedestrian}(x)$, we get \mathbf{h} as the only answer, as \mathbf{h} is a driver in \mathcal{D}_0^{ex} and the ontology states that drivers can never be pedestrians. However, if we change the query to $\Phi_2^{ex} = \Box \neg \mathbf{Pedestrian}(x)$, no individual satisfies the constraint, since \mathcal{D}_1^{ex} asserts nothing – it can very well be possible that \mathbf{h} has become a pedestrian (due to the OWA).

This example highlights that languages like MTCQs are important for testing requirements on systems in complex domains. However, up to now, only the theoretical work by Baader et al. examines a related but hard-to-implement setting over infinite traces for complexity-theoretic analyses [8]. No language

has yet been defined that is practically suitable for implementation and has the required expressiveness. Moreover, there currently is no tooling for *any* temporal query language over expressive DLs. Our work on MTCQs addresses this gap.

For this, we first introduce the formal foundation of MTCQs in Section 3. We implement the framework in an answering engine for a large and practically relevant subclass of MTCQs in Section 4, closing the identified research gap. To evaluate its efficacy, we present a benchmark generator for temporal knowledge bases, as described in Section 5. We show the efficacy of our tool in this practical setting in Section 6. To summarize, the main contributions of our work are

1. MTCQs as a practically implementable and expressive temporal query language and the first tool for answering such queries up to the DL $\mathcal{SRIQ}^{(D)}$,
2. a benchmark generator for the evaluation of inference tasks on temporal knowledge bases, and
3. an application of the tool in our motivational setting of situation recognition for urban automated driving.

2 Related Work

We previously claimed that for our motivational domain of ADS development the usefulness of temporal logics (TLs) and related mechanisms – e.g., regular expressions – for scenario extraction has been recognized, which is supported by the literature [26, 31, 18, 16]. More specifically, work exists in specifying behavioral requirements, e.g., based on traffic rules, using TLs [1, 33, 19]. However, none of these approaches formally incorporate an ontology. In general, the importance of ontologies in automated driving is recognized, see, e.g., ASAM OpenXOntology [7] for an international standardization project as well as Westhofen et al. [42] and Zipfl et al. [44] for non-systematic reviews. Some ontological approaches are in fact based on DLs [27]. However, we are not aware of work within the automotive domain that uses DLs and TLs for analyzing temporal traffic data.

On the theoretical side, a plethora of temporal DLs have been introduced [2, 32, 5], also on finite traces [6]. These classical combinations were not conceived in a query answering context, so more recently, several frameworks for addressing that have been introduced [3]. We mention the most important ones here. There is work on ontologies formulated in the lightweight (i.e., comparatively inexpressive) DLs DL-Lite [12, 38] and \mathcal{EL} [13, 22]. For expressive DLs, an important line of work theoretically examines answering temporal conjunctive queries – essentially infinite-time LTL over conjunctive queries – over temporal knowledge bases with the ontology language ranging from \mathcal{ALC} [8] to \mathcal{SHQ} [30, 9]. Related, but orthogonal to combinations of DLs with TLs, are combinations of Datalog with TLs. This line of research started around 1990 with Datalog1S [15], and lead to other combinations [14, 39] for which also tools exist [40].

3 Formal Foundations

We introduce the formal foundations of the relevant DLs and their temporal extension. For the sake of simplicity, we focus on the ontology language \mathcal{ALC} , which is a prototypical language in the class of expressive DLs. However, our approach generalizes to (and is actually implemented for) the more expressive logic $\mathcal{SRIQ}^{(D)}$, cf. Horrocks et al. for further reference on this DL fragment [24].

We start with an introduction to non-temporal knowledge bases which we later use as a foundation for defining the temporal case. As sketched in Section 1, in \mathcal{ALC} we can describe the relationship of roles and concepts in an ontology \mathcal{O} and assert individuals to these concepts and roles in a database \mathcal{D} . Any knowledge base is thus a tuple $(\mathcal{O}, \mathcal{D})$ and relies upon concept, role, and individual names. For the remainder, we fix countably infinite supplies N_C, N_R, N_I of concept, role, and individual names, respectively. An \mathcal{ALC} -concept description C is formed according to $C ::= A \mid \neg C \mid C \sqcap C \mid C \sqcup C \mid \forall r.C \mid \exists r.C$ where A ranges over N_C and r ranges over N_R . We can thus compose new concepts using negation, intersection, and union. For a role r , we moreover allow for universal (enforcing a concept to only have r -successors in C) and existential quantification (enforcing a concept to have an r -successor in C). Section 1 already introduced an example of an existentially quantified role using $\exists \text{drives.Vehicle}$ – the concept of all things driving some vehicle. An ontology is a set of concept inclusions $C \sqsubseteq D$ for \mathcal{ALC} -concepts C and D , denoting subsumption of the concept C to the concept D . We write $C \equiv D$ (concept equivalence) for $C \sqsubseteq D$ and $D \sqsubseteq C$. Again, the introduction used $\text{Human} \sqcap \exists \text{drives.Vehicle} \equiv \text{Driver}$ as an example for concept equivalence. The data is a set of facts of the form $A(a)$ and $r(a, b)$ for $a, b \in N_I$, $r \in N_R$, and $A \in N_C$, hence assigning individuals to concepts and roles. We denote the set of individuals that occur in \mathcal{D} by $\text{Ind}(\mathcal{D})$. The introductory example of Section 1 used the set of individuals $\{h, v\}$ and asserted the role $\text{drives}(h, v)$.

The semantics of ontologies and data is defined via interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a domain $\Delta^{\mathcal{I}}$ and a mapping $\cdot^{\mathcal{I}}$ that assigns a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to every $A \in N_C$, a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to every role name $r \in N_R$, and an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to every $a \in N_I$ [10, Chapter 2.2]. As to incorporate \mathcal{ALC} -concept descriptions, the interpretation function is inductively defined as:

$$\begin{aligned} (\neg C)^{\mathcal{I}} &:= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\ (C \sqcap D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\ (C \sqcup D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\ (\forall r.C)^{\mathcal{I}} &:= \{c \in \Delta^{\mathcal{I}} \mid \forall d \in \Delta^{\mathcal{I}}. (c, d) \in r^{\mathcal{I}} \rightarrow d \in C^{\mathcal{I}}\} \\ (\exists r.C)^{\mathcal{I}} &:= \{c \in \Delta^{\mathcal{I}} \mid \exists d \in \Delta^{\mathcal{I}}. (c, d) \in r^{\mathcal{I}} \wedge d \in C^{\mathcal{I}}\} \end{aligned}$$

Then, we say $\mathcal{I} \models C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, $\mathcal{I} \models A(a)$ if $a^{\mathcal{I}} \in A^{\mathcal{I}}$, and $\mathcal{I} \models r(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$. As to lift these definitions to ontologies and data, we write $\mathcal{I} \models \mathcal{O}$ and $\mathcal{I} \models \mathcal{D}$ if \mathcal{O} resp. \mathcal{D} satisfy *all* concept inclusions in \mathcal{O} resp. assertions in \mathcal{D} . Finally, for a complete knowledge base, we define $\mathcal{I} \models (\mathcal{O}, \mathcal{D})$ if $\mathcal{I} \models \mathcal{O}$ and $\mathcal{I} \models \mathcal{D}$. More details on the semantics of DLs are given by Baader et al. [10].

We now extend this definition of non-temporal knowledge bases to the temporal case, where a knowledge base consists of an ontology \mathcal{O} and a finite sequence of assertions that describe the databases over time.

Definition 1 (Temporal Knowledge Base). A temporal knowledge base (KB) is a tuple $\mathcal{K} = (\mathcal{O}, (\mathcal{D}_i)_{i \in \{0, \dots, n\}})$ where \mathcal{O} is an ontology and each \mathcal{D}_i is a database.

Their semantics is defined by temporal interpretations using the non-temporal case as its basis.

Definition 2 (Temporal Interpretation). A temporal interpretation \mathcal{I} is a finite sequence $\mathcal{I} = (\mathcal{I}_i)_{i \in \{0, \dots, m\}}$ of interpretations over a fixed domain Δ such that $a^{\mathcal{I}_i} = a^{\mathcal{I}_j}$, for all $a \in \mathbf{N}_I$ and $0 \leq i, j \leq m$. We call \mathcal{I} a model of the temporal KB $(\mathcal{O}, (\mathcal{D}_i)_{i \in \{0, \dots, n\}})$, written $\mathcal{I} \models \mathcal{K}$, if $m = n$ and $\mathcal{I}_i \models \mathcal{D}_i$ and $\mathcal{I}_i \models \mathcal{O}$, for all $i \in \{0, \dots, n\}$.

The assumption that all interpretations share a common domain is called *constant domain assumption*. We define next the language MTCQ that we use to query temporal KBs. It is a combination of standard conjunctive queries with temporal operators inspired by MLTL [29].

Definition 3 (Syntax of MTCQs). Let \mathbf{N}_V be a countably infinite set of variable names. A conjunctive query (CQ) φ is an expression of the form $\varphi(\vec{x}) = \exists \vec{y}. \psi(\vec{x}, \vec{y})$ where \vec{x}, \vec{y} are tuples of variables from \mathbf{N}_V and ψ is a conjunction of concept atoms $A(t)$ and role atoms $r(t, t')$ with $A \in \mathbf{N}_C$, $r \in \mathbf{N}_R$, and $t, t' \in \vec{x} \cup \vec{y} \cup \mathbf{N}_I$. Metric temporal conjunctive queries (MTCQs) Φ are built from CQs using negation $\neg\Phi$, conjunction $\Phi \wedge \Phi'$, and two versions of until, $\Phi \mathcal{U} \Phi'$ and $\Phi \mathcal{U}_{[a,b]} \Phi'$ for $a, b \in \mathbb{N}$. We denote with $\text{Ind}(\Phi)$ the set of individuals and $\text{Var}(\Phi)$ the set of variables in an MTCQ Φ .

Note that we extend MLTL by borrowing the unconstrained until operator from LTL, because it is a frequent operator in practice. Additionally, it allows for a more direct translation to finite automata in our system presented later on. We call the variables \vec{x} the *answer variables* and \vec{y} the *quantified variables*. An MTCQ is *Boolean* if it does not have answer variables. The semantics of Boolean CQs is defined in terms of matches into interpretations.

Definition 4 (Semantics of Boolean CQs). For a Boolean conjunctive query φ and an interpretation \mathcal{I} , $\mathcal{I} \models \varphi$ iff there exists a function $\pi: \text{Var}(\varphi) \cup \text{Ind}(\varphi) \rightarrow \Delta^{\mathcal{I}}$ with 1. $\pi(a) = a^{\mathcal{I}}$ for all $a \in \text{Ind}(\varphi)$, 2. $\pi(t) \in \mathcal{C}^{\mathcal{I}}$ for all $\mathcal{C}(t)$ in φ , and 3. $(\pi(t), \pi(t')) \in \mathbf{r}^{\mathcal{I}}$ for all $\mathbf{r}(t, t')$ in φ .

Hence, an interpretation satisfies a Boolean CQ if the interpretation can respect its constraints. Boolean CQs form the basis for the semantics of Boolean MTCQs.

Definition 5 (Semantics of Boolean MTCQs). Let $\mathcal{I} = (\mathcal{I}_i)_{i \in \{0, \dots, m\}}$ be a temporal interpretation and $i \in \{0, \dots, m\}$. The semantics of Boolean MTCQs is given by structural induction:

- $\mathcal{I}, i \models \Phi$ iff $\mathcal{I}_i \models \Phi$, if Φ is a Boolean CQ;

- $\mathcal{I}, i \models \neg\Phi$ iff $\mathcal{I}, i \not\models \Phi$;
- $\mathcal{I}, i \models \Phi_1 \wedge \Phi_2$ iff $\mathcal{I}, i \models \Phi_1$ and $\mathcal{I}, i \models \Phi_2$;
- $\mathcal{I}, i \models \Phi_1 \mathcal{U}_{[a,b]} \Phi_2$ iff there is a $k \in [a, b]$ with $i+k \leq m$ such that $\mathcal{I}, i+k \models \Phi_2$ and $\mathcal{I}, i+j \models \Phi_1$, for all $j \in [a, k]$;
- $\mathcal{I}, i \models \Phi_1 \mathcal{U} \Phi_2$ iff there is a $k \in [i, m]$ such that $\mathcal{I}, k \models \Phi_2$ and $\mathcal{I}, j \models \Phi_1$, for all $j \in [i, i+k]$.

We allow the typical abbreviations $\Phi \vee \Phi'$ for $\neg(\neg\Phi \wedge \neg\Phi')$, **false** for $\exists x.A(x) \wedge \neg\exists x.A(x)$ for some $A \in \mathbf{N}_C$, **true** for $\neg\mathbf{false}$, $\diamond_{[a,b]}\Phi$ for $\mathbf{true}\mathcal{U}_{[a,b]}\Phi$, $\diamond\Phi$ for $\mathbf{true}\mathcal{U}\Phi$, $\square_{[a,b]}\Phi$ for $\neg\diamond_{[a,b]}\neg\Phi$, and $\square\Phi$ for $\neg\diamond\neg\Phi$. The *strong next-operator* is defined as $\bigcirc\Phi \equiv \diamond_{[1,1]}\Phi$ and *weak next* as $\bullet\Phi \equiv \square_{[1,1]}\Phi$. Note that finite trace semantics exhibit some non-obvious behaviors, e.g., $\diamond\square\Phi$ is equivalent to $\square\diamond\Phi$ [17].

A central problem over Boolean MTCQs is *entailment*: For a temporal KB \mathcal{K} and an MTCQ Φ , we say $\mathcal{K} \models \Phi$ if for all temporal interpretations \mathcal{I} with $\mathcal{I} \models \mathcal{K}$ also $\mathcal{I}, 0 \models \Phi$ holds. For example, for \mathcal{K}^{ex} from Section 1, it holds that $\mathcal{K}^{ex} \models \diamond\neg\mathbf{Pedestrian}(\mathbf{h})$ as for any temporal interpretation $\mathcal{I} = (\mathcal{I}_0, \mathcal{I}_1)$ with $\mathcal{I}_0 \models \mathcal{O}^{ex}$ and $\mathcal{I}_0 \models \mathcal{D}_0^{ex}$, it must also hold that $\mathbf{h}^{\mathcal{I}_0} \in (\neg\mathbf{Pedestrian})^{\mathcal{I}_0}$ due to the fact that \mathbf{h} is inferred to be a driver and thus cannot be a pedestrian.

We remark that this semantics is closely related to the one over temporal conjunctive queries (TCQs) introduced by Baader et al. [8] to query temporal KBs over arbitrary models, i.e., not restricted to mission time. In fact, it is not difficult to see that entailment $\mathcal{K} \models \Phi$ for Boolean MTCQs Φ can be reduced to deciding whether \mathcal{K} entails $\widehat{\Phi}$ in the sense of Baader et al. [8] for some TCQ $\widehat{\Phi}$ that can be computed in polynomial time from Φ ; we denote the latter entailment relation with $\mathcal{K} \models^{\text{BBL}} \widehat{\Phi}$. In the mentioned paper it is also shown that the latter entailment problem is in ExpTime. Together with the ExpTime-lower bound for subsumption in \mathcal{ALC} this shows that MTCQ entailment is ExpTime-complete. Of course, the complexity is potentially higher for ontology languages beyond \mathcal{ALC} . Finally, if in place of CQs in MTCQs we allow for \mathcal{ALC} -concepts, the resulting language can be embedded into the metric temporal DLs discussed by Gutiérrez-Basulto et al. [23].

While Boolean MTCQ entailment is the natural problem to consider for complexity analysis, a practical system needs support for *answering non-Boolean MTCQs*, which is defined based on entailment. Let $\mathcal{K} = (\mathcal{O}, (\mathcal{D}_i)_{i \in \{0, \dots, n\}})$ be a temporal KB, $\Phi(\vec{x})$ an MTCQ with answer variables \vec{x} , and \vec{a} a tuple of individuals from \mathcal{K} , i.e., $\vec{a} \subseteq \text{Ind}(\mathcal{K}) := \bigcup_{i=0, \dots, n} \text{Ind}(\mathcal{D}_i)$. We call \vec{a} a *certain answer to $\Phi(\vec{x})$ over \mathcal{K}* if $\mathcal{K} \models \Phi(\vec{a})$. Here, $\Phi(\vec{a})$ is the uniform replacement of the variables in \vec{x} by the individual names in \vec{a} , leading to a Boolean MTCQ. Our main reasoning task is to compute the set $\text{cert}_{\mathcal{K}}(\Phi)$ of certain answers of Φ over \mathcal{K} . Section 1 gives an example for this set: $\text{cert}_{\mathcal{K}^{ex}}(\diamond\neg\mathbf{Pedestrian}(x)) = \{\mathbf{h}\}$.

4 Computing Certain Answers in Practice

We start with noting that to compute $\text{cert}_{\mathcal{K}}(\Phi)$, it is not sufficient to answer all of Φ 's CQs at time i and combine them inductively according to the semantics

due to the presence of disjunction in our query language. An example is the MTCQ $\Phi_{\vee}(x) := B(x) \vee C(x)$ over the temporal KB $\mathcal{K}_{\vee} := (A \sqsubseteq B \sqcup C, (A(a)))$, where $\text{cert}_{\mathcal{K}_{\vee}}(\Phi_{\vee}) = \{(a)\}$. A separate check of $B(x)$ and $C(x)$ returns no answer, and inductive combination falsely yields no answer as well. This issue explains the restriction to conjunctions in existing CQ answering implementations over expressive DLs, as complexity is reduced and various optimizations can be employed. Therefore, and in contrast to both LTL_f over propositional atoms and CQ answering, we require a more involved procedure for checking MTCQs.

The correct but naïve way to compute $\text{cert}_{\mathcal{K}}(\Phi)$ is to enumerate all candidate answers $\vec{a} \subseteq \text{Ind}(\mathcal{K})$ and decide whether $\mathcal{K} \models^{\text{BBL}} \Phi(\vec{a})$ via the algorithms provided by Baader et al. [8] (for the temporal aspects) and Horrocks and Tessaris [25] (for answering disjunctions of conjunctive queries). This, however, suffers from several problems. First, there are potentially many answer candidates since the number of relevant tuples is exponential in the arity of the query Φ . Second, while the mentioned algorithm for deciding \models^{BBL} is useful for a complexity analysis, it does not lend itself to a direct implementation. Finally, the algorithm of Baader et al. works over unrestricted models and is thus more difficult to implement. This section provides the foundations for the algorithm that we implemented in our tool and the central improvements needed to make it work in practice.

As MTCQs are closed under negation, entailment is just the complement of *satisfiability*: a Boolean MTCQ Φ is satisfiable w.r.t. a temporal KB \mathcal{K} if there is a model \mathcal{I} of \mathcal{K} with $\mathcal{I}, 0 \models \Phi$. As $\mathcal{K} \models \Phi$ iff $\neg\Phi$ is unsatisfiable w.r.t. \mathcal{K} , we can, for the sake of convenience, focus on satisfiability in the following.

We need some preliminary notions. Given an MTCQ Φ (possibly with answer variables), we denote with $\text{CQ}(\Phi)$ the set of all CQs in Φ . The *propositional abstraction* $\text{PA}(\Phi)$ of Φ is the replacement of each $\varphi \in \text{CQ}(\Phi)$ with a propositional variable p_{φ} . Note that the propositional abstraction of an MTCQ is an MLTL formula potentially with an unconstrained until, which is the underlying temporal formalism. This TL is interpreted over finite words $P_0 \cdots P_n$ where each P_i specifies the propositional variables that are satisfied at time point i . Boolean operators are interpreted as usual and temporal operators \mathcal{U} and $\mathcal{U}_{[a,b]}$ are interpreted in line with Definition 5. The following characterization of satisfiability is easy to prove from the definitions.

Lemma 1. *For a Boolean MTCQ Φ and a temporal KB $\mathcal{K} = (\mathcal{O}, (\mathcal{D}_i)_{i \in \{0, \dots, n\}})$, Φ is satisfiable w.r.t. \mathcal{K} iff there is a sequence X_0, \dots, X_n of subsets of $\text{CQ}(\Phi)$ such that:*

1. *there are interpretations $\mathcal{I}_0, \dots, \mathcal{I}_n$ over the same domain such that, for all $i \in \{0, \dots, n\}$, we have $\mathcal{I}_i \models \mathcal{O}$, $\mathcal{I}_i \models \mathcal{D}_i$, and $\mathcal{I}_i \models \varphi$ for every $\varphi \in X_i$, and $\mathcal{I}_i \not\models \varphi$ for every $\varphi \in \text{CQ}(\Phi) \setminus X_i$, and*
2. *$(\{p_{\varphi} \mid \varphi \in X_i\})_{i \in \{0, \dots, n\}}$ satisfies $\text{PA}(\Phi)$.*

Intuitively, Lemma 1 splits the problem of deciding MTCQ satisfiability into separate DL and TL tasks which are only connected by the sets of CQs X_0, \dots, X_n .

Lemma 1 can be further refined as follows. The requirement that all interpretations $\mathcal{I}_0, \dots, \mathcal{I}_n$ be over the same domain can be dropped without compromising correctness. Indeed, we can combine $\mathcal{I}_0, \dots, \mathcal{I}_n$ witnessing Point 1 in Lemma 1 but with potentially different domains into $\mathcal{I}'_0, \dots, \mathcal{I}'_n$ with the same domain using a standard argument, cf. the proof of Theorem 5.21 by Lippmann [30]: Since \mathcal{ALC} cannot enforce finite models, we can assume that each \mathcal{I}_i is infinite. By the downward Löwenheim-Skolem-Theorem, we can assume that the \mathcal{I}_i are countably infinite and thus have the same domain. It remains to identify the interpretation of the individual names. Note that the argument goes through for more expressive logics such as $\mathcal{SRIQ}^{(\mathcal{D})}$.

Lemma 2. *Lemma 1 remains valid when “over the same domain” is dropped from Point 1.*

Hence, the checks at each time in (the modified) Point 1 are independent. It remains to show how we can implement the check of Point 1, which includes negated CQs. By the natural connection between satisfiability and entailment, we can leverage an engine for answering disjunctions of CQs over non-temporal \mathcal{ALC} KBs for this, i.e., computing $\text{cert}_{\mathcal{K}}(\Phi)$ for $\mathcal{K} = (\mathcal{O}, \mathcal{D})$ and Φ a disjunction of CQs. For doing so, we associate with every Boolean CQ φ its *canonical database* \mathcal{D}_{φ} which is just the set of all conjuncts that occur in φ . (For the sake of simplicity, we allow variable names from φ as individual names in \mathcal{D}_{φ} .) We then exploit the following observation.

Observation 1 *Let X be a set of Boolean CQs, let \mathcal{O} be an \mathcal{ALC} -ontology and \mathcal{D} the data. Then the following are equivalent for every subset $Z \subseteq X$:*

- (a) *There is a model \mathcal{I} of \mathcal{O} and \mathcal{D} such that $\mathcal{I} \models \varphi$ for every $\varphi \in Z$, and $\mathcal{I} \not\models \varphi$ for every $\varphi \in X \setminus Z$.*
- (b) *$(\mathcal{O}, \mathcal{D}') \not\models \bigvee_{\varphi \in X \setminus Z} \varphi$ where \mathcal{D}' is the union of \mathcal{D} with \mathcal{D}_{φ} for each $\varphi \in Z$ (with variables across different \mathcal{D}_{φ} suitably renamed).*

Thus, to check the modified Point 1 for some time point (a condition of shape (a) in the above Lemma), we can check its reformulation as (b) using a (non-temporal) query engine for disjunctions of CQs. As demonstrated by the exemplary query $\Phi_{\vee}(x)$, this is, however, more involved than answering each disjunction separately, a problem already known to the DL community. For correctly answering such disjunctions of CQs, we require a reformulation in of the disjunction into conjunctive normal form, and then answer each conjunct separately as described by Horrocks et al. [25]. For $P \subseteq \{p_{\varphi} \mid \varphi \in \text{CQ}(\Phi)\}$, we define $\text{VAL}_{\Phi}^i(P)$ as true iff. $\mathcal{O}, \mathcal{D} := \mathcal{D}_i, Z := \{\varphi \mid p_{\varphi} \in P\}, X := \text{CQ}(\Phi)$ pass the test in Point (b), and thus the modified Point 1.

To implement Point 2, we exploit that for each MLTL formula χ over some set of propositions Σ , one can compute an equivalent LTL_f (LTL over finite traces) formula χ' over Σ [29] which in turn can be transformed into a finite automaton (FA) \mathfrak{A}_{χ} over 2^{Σ} which recognizes precisely the models of χ' and thus of χ [17]. Both these transformations are not polynomial and there is, in general, no efficient conversion of an MLTL formula to an FA. However, since

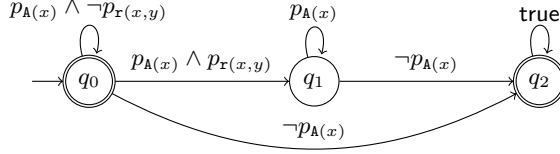


Fig. 1. FA for $PA(\neg(\Box A(x) \wedge \Diamond r(x, y)))$.

Algorithm 1 Computing certain answers to MTCQs.

Input: MTCQ $\Phi(\vec{x})$, temporal KB $\mathcal{K} = (\mathcal{O}, (\mathcal{D})_{i \in \{0, \dots, n\}})$

Output: $\text{cert}_{\mathcal{K}}(\Phi)$.

```

1:  $\mathfrak{D} := \text{CONSTRUCT\_FA}(PA(\neg\Phi))$ ;
2: // states  $Q$ , initial state  $q_0$ , final states  $F$ , transitions  $\Delta$ 
3:  $C := \text{ind}(\mathcal{D})^k$  where  $k = |\vec{x}|$ 
4: Initialize  $S(\vec{a}, 0) := \{q_0\}$  for all  $\vec{a} \in C$ 
5: for  $i := 1$  to  $n + 1$  do
6:   for  $\vec{a} \in C$  do
7:      $S(\vec{a}, i) := \emptyset$ 
8:     for  $q \in S(\vec{a}, i - 1)$  do
9:        $S(\vec{a}, i) := S(\vec{a}, i) \cup \{q' \mid (q, X, q') \in \Delta, \text{VAL}_{\neg\Phi(\vec{a})}^{i-1}(X)\}$ 
10:    end for
11:  end for
12: end for
13: return  $\{\vec{a} \in C \mid S(\vec{a}, n + 1) \cap F = \emptyset\}$ ;
```

queries are often small in practice, this is still feasible. For example, the minimal FA for $p_1 \mathcal{U}_{\leq a} \Diamond_{\leq b} p_2$ has $a + b + 3$ states. Figure 1 shows the FA for answering the simple MTCQ $\Phi_{ex}(x, y) = \Box A(x) \wedge \Diamond r(x, y)$. Note that the transitions are labeled with Boolean formulas over the propositions indicating a transition for each model of the formula, which can be exponentially more succinct.

What was said so far suggests the basic procedure for computing $\text{cert}_{\mathcal{K}}(\Phi)$ that is depicted in Algorithm 1. It considers for each answer candidate \vec{a} all possible 'runs' X_0, \dots, X_n in a step-by-step fashion and checks (modified) Points 1 and 2 after each step; the set $S(\vec{a}, i)$ contains all states the FA corresponding to $\neg\Phi(\vec{a})$ can reach after i steps. The central test happens in Line 7 and is given here for the direct encoding of the transitions; it can easily be adapted for the mentioned succinct encoding. The algorithm returns all \vec{a} for which no final state is reachable after $n + 1$ steps. Applied to the example FA in Figure 1 and a candidate answer (a, b) this means that the FA ends up in state q_1 in all possible runs, according to the temporal KB. The only way to achieve this is for the FA to not stay in q_0 or q_2 . For this, it has to eventually change from q_0 to q_1 by having neither $A(a) \wedge \neg r(a, b)$ nor $\neg A(a)$ but $A(a) \wedge r(a, b)$ satisfiable. The FA shall then stay solely in q_1 with only $A(a)$ satisfiable for the remainder. Clearly, in this case (a, b) is a certain answer.

4.1 Improvements

Some standard improvements over Algorithm 1 are applicable, e.g., to work directly on a minimal FA. However, this does not yet address the problem of the many answer candidates to consider, of which, in practice, only few will be entailed. Algorithm 1 considers each candidate individually, which is inefficient since similar tasks are repeatedly executed. We instead leverage existing systems that implement efficient algorithms specifically tailored towards answering CQs over standard (non-temporal) KBs. As an example, consider again the FA in Figure 1. Observe that $q_2 \in S(\vec{a}, i)$ for all \vec{a}, i for which $(\mathcal{O}, \mathcal{D}_{i-1}) \not\models \mathbf{A}(\vec{a})$. Indeed, $\neg \mathbf{A}(\vec{a})$ is satisfiable w.r.t. $(\mathcal{O}, \mathcal{D}_{i-1})$, for those \vec{a}, i . Since q_2 is a sink, this allows us to instantly reject all non-answers to $\mathbf{A}(x)$. We now generalize this to extract certain (non-)answers by answering the CQs occurring in the edges.

The main idea is to perform an under-approximating traversal of the FA prior to Algorithm 1. More concretely, we use CQ answering to construct sets $R(\vec{a}, i) \subseteq S(\vec{a}, i)$ and $U(\vec{a}, i) \subseteq Q \setminus S(\vec{a}, i)$ that under-approximate the reachable and unreachable states, respectively, for a candidate \vec{a} at time i . This serves two purposes. First, we can already extract some certain answers from U and some certain non-answers from R , namely the sets $\{\vec{a} \in C \mid U(\vec{a}, n+1) \supseteq F\}$ and $\{\vec{a} \in C \mid R(\vec{a}, n+1) \subseteq F\}$, respectively. These candidates are not considered anymore during the run of Algorithm 1. Second, we are able to re-use cached answers to CQs in the first traversal during Algorithm 1.

We now describe how to construct the sets R and U during FA traversal. $R(\vec{a}, 0)$ is initialized as $\{q_0\}$ and $U(\vec{a}, 0)$ is initialized as $Q \setminus \{q_0\}$, for all \vec{a} . For the update step with $i > 0$, we assume for all states q_k, q_l to have succinctly encoded edges $\alpha_{k,l} := \bigwedge_{p_\varphi \in P_0} \neg p_\varphi \wedge \bigwedge_{p_\varphi \in P_1} p_\varphi$ for some sets $P_0, P_1 \subseteq P$, as already used in Figure 1. When examining such an edge in the FA at time i , we use a CQ engine on $\mathcal{K}_i := (\mathcal{O}, \mathcal{D}_i)$ to compute $\text{cert}_{\mathcal{K}_i}(\varphi)$ for all $\varphi \in \{\psi \mid p_\psi \in P_0\} \cup \{\bigwedge_{p_\psi \in P_1} \psi\}$. From these sets, we are able to extract information on the relevant queries:

1. for all $\vec{a} \notin \text{cert}_{\mathcal{K}_i}(\varphi)$: $\neg\varphi(\vec{a})$ is *satisfiable* w.r.t. \mathcal{K}_i ;
2. for all $\vec{a} \in \text{cert}_{\mathcal{K}_i}(\varphi)$: $\varphi(\vec{a})$ is *satisfiable* and $\neg\varphi(\vec{a})$ is *unsatisfiable* w.r.t. \mathcal{K}_i .

We transfer this knowledge about the (un-)satisfiability of $\varphi(\vec{a})$ and $\neg\varphi(\vec{a})$ to the edges $\alpha_{k,l}$. *Satisfiability* knowledge is transferable if $q_k \in R(\vec{a}, i-1)$ and $\alpha_{k,l} = p_\varphi$ resp. $\alpha_{k,l} = \neg p_\varphi$. We then add q_l to $R(\vec{a}, i)$. *Unsatisfiability* knowledge on $\neg\varphi(\vec{a})$ is transferable if $\alpha_{k,l}$ contains $\neg p_\varphi$. Adding unsatisfiability knowledge to U requires adaptations. Firstly, we can only add q_l to $U(\vec{a}, i)$ if *all* other edges $\alpha_{j,l}$ to q_l also agree on unsatisfiability of \vec{a} at time i , i.e., they contain some $\neg p_{\varphi'}$ for which $\varphi'(\vec{a})$ is known to be unsatisfiable or $q_j \in U(\vec{a}, i-1)$. Secondly, unsatisfiability generates new satisfiability information: for a state q_k with successors q_{l_1}, \dots, q_{l_h} we know that $\{q_{l_1}, \dots, q_{l_{h-1}}\} \subseteq U(\vec{a}, i)$ implies $q_{l_h} \in R(\vec{a}, i)$. Together with the described acceptance condition, the sets $R(\vec{a}, n+1)$ and $U(\vec{a}, n+1)$ deliver an under-approximation of the certain (non-)answers.

4.2 Our System

We implemented this approach as a module in the DL reasoner OPENLLET [37]. The implementation is available at <https://github.com/lu-w/topllet>. Our module does not support full MTCQs yet. Instead of allowing arbitrary CQs as atoms, we allow the subclass tCQ of CQ which consists of all CQs φ s.t. in the graph $G_\varphi = (V, E)$ with $V = \text{Var}(\varphi) \cup \text{Ind}(\varphi)$ and $E = \{(t, r, t') \mid r(t, t') \in \varphi\}$ each vertex has at most one incoming edge and, if interpreted undirectedly, G is acyclic, i.e., the query graph is *tree-shaped*⁴.

We denote with tMTCQ the subclass of MTCQ where each CQ is in fact a tCQ. The reasons for considering this query class are two-fold. First, most queries that occur in practice are tMTCQs. Second, tCQ answering can be implemented by a straightforward procedure of ‘rolling-up’ the query graph [25]. Therefore, OPENLLET already provides an tCQ-answering engine over $\mathcal{SROIQ}^{(D)}$ KBs, implementing many optimizations [36]. Moreover, the procedure can be adapted to answering disjunctions of tCQs as described by Horrocks and Tessaris [25], which required for our algorithm, cf. Point (b) in Observation 1.

As a first necessary step, we thus extended OPENLLET to being able to answer disjunctions of tCQs. For the construction of the FA, we implemented the conversion of MLTL to LTL_f described by Li et al. [29]: essentially, the intervals in $\mathcal{U}_{[a,b]}$ are encoded using sequences of the next-operator \bigcirc of length a and b , respectively. We then rely on LYDIA, which converts LTL_f formulas to equivalent deterministic FA [20]. We extend and use the AUTOMATALIB [28] to access the resulting FA. We provide a test suite for our system to highlight correctness of the implemented algorithms.

5 Benchmarks

Our CQ answering approach motivates the need for empirical evaluation, for which ideally controlled real-world data is used. In fact, for one experiment, we obtained drone data from an intersection in Germany. These data turned out to be insufficient for a thorough evaluation, as they are proprietary and not scalable. This calls for synthetic yet realistic benchmark data that can be randomized, scaled in size, and are freely available for replicability. However, we are currently not aware of *any* public benchmark data on querying temporal KBs. The same was noted by the developers of METEOR, where data of the Lehigh University Benchmark [21] are extended with random intervals to enable an evaluation on the OWL RL fragment of LUBM. Unfortunately, a random extension of a non-temporal benchmark might not reflect actual temporal data, e.g., in continuity of concepts over time, and thus might not transfer to real-world applications. As our final contribution, we hence present the Traffic Ontology Benchmark (TOBM), a benchmark generator for scenarios of automated driving applications that mimics

⁴ This constraint allows us to perform the rolling-up procedure on the BCQs of the FA. However, it is actually just a sufficient condition for rolling-up. More precisely, we require the FA to contain only BCQs where each negated query is a tCQ.

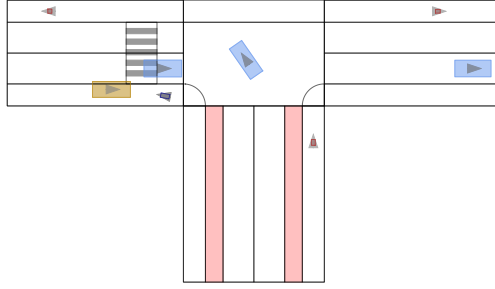


Fig. 2. A scene of the T-crossing scenario sampled from TOBM.

real-world data and enables to evaluate tools on temporal KBs, including MTCQ answering. The tool is available at <https://github.com/lu-w/tobm>.

For the ontology we rely on the publicly available Automotive Urban Traffic Ontology (A.U.T.O.) [42, Section 5]. It is a conglomerate of $SRIQ^{(D)}$ ontologies for the traffic domain and related fields, and currently consists of 1449 axioms over 676 concepts and 213 roles. A.U.T.O. was already successfully used for analyzing real-world traffic data from drone recordings [42, Section 8].

The benchmark generator creates temporal data for A.U.T.O. with individuals scaling linear to some $N > 0$. A seed S can be used for pseudo-randomization. From both parameters, it generates scenarios of a certain length (by default, 20 seconds). These can be sampled from two settings:

1. A T-crossing setup with parking vehicles, a pedestrian crossing, bikeway lanes, pedestrians, bicyclists, and passenger cars (cf. Figure 2). It has $8 \cdot N + 22$ individuals.
2. An X-crossing of two urban roads with traffic signs and dysfunctional traffic lights. Compared to the T-crossing, there are no bicyclists and $5 \cdot N + 69$ individuals.

The scenarios are created based on behavior models for pedestrians, bicyclists, and passenger cars. Passenger cars and bicyclists drive up to a speed limit if their front area is free, otherwise they use a following mode. Vehicles yield on a predicted intersecting path. Moreover, a random successor lane is selected when turning at intersections, giving a turning signal with a probability of 3% each time point. Pedestrians follow their walkway, but can randomly initiate road crossing with a probability of 0.7%. We give a visualization of two exemplary scenarios can be found in the linked repository.

Our implementation models temporal KBs as a list of OWL2-files for the data, each importing a shared ontology. Geometrical data are abstracted to spatial predicates (e.g., `is_in_front_of`) in a pre-processing step. For $S = 0$, $N = 3$, and 20 seconds sampled with 10 Hertz on the T-crossing setting, this results in a data sequence with 46 individuals and 647847 assertions in total (approx. 3239 per time point) with constant assertions only counted once.

6 Evaluation

We now examine practical feasibility of our system by an evaluation on TOBM, answering the following questions:

1. Is the approach applicable to practical, a-posteriori situation recognition tasks (such as evaluating test data) with larger numbers of assertions?
2. What is the impact of our improvement of leveraging CQ answering on overall applicability?
3. In practical settings, how much satisfiability knowledge can be generated by CQ answering?

As inputs, we sampled TOBM with $S = 0$ and $N \in \{1, \dots, 5\}$ for both the X- and T-crossing. We fix a 20 second duration with ten Hertz, as our algorithm performs linear in N . The supplementary artifact provides both the benchmarks and a wrapper around TOBM for reproducible re-generation. We used four queries (given in the supplementary artifact) asking for: intersecting paths with VRUs (Φ_1), passing of parking vehicles on two-lane roads (Φ_2), vehicles turning right (Φ_3), and vehicles changing lanes without signals (Φ_4), where Φ_1 , Φ_2 , and Φ_3 have two and Φ_4 has three answer variables. The corresponding FAs have 8 (Φ_1), 4 (Φ_2, Φ_4), and 3 (Φ_3) states. Our tool is executed once per benchmark and query combination, as deviations are not be expected due to determinism, on an Intel Core i9-13900K with 64 GB RAM and a time limit of ten hours per run, using a Windows Subsystem for Linux 1 on a Windows 10 host. The input files and tool, with the exact version and configuration used for benchmarking, are available online [41].

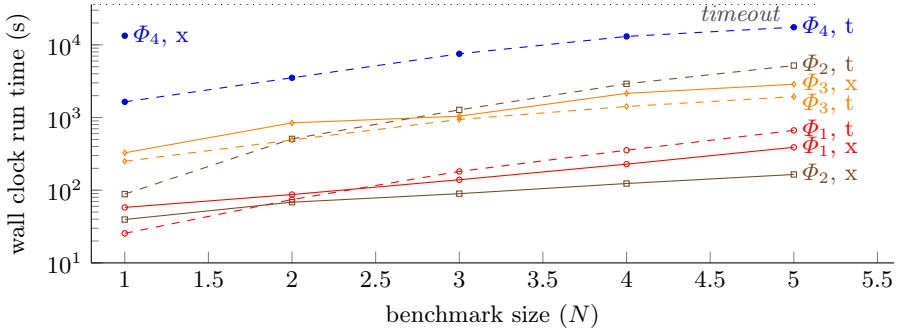


Fig. 3. Wall clock running times of benchmark queries Φ_i , $i \in \{1, \dots, 4\}$ and the T- (t) resp. X-crossing (x) of size N .

For the first question, we show wall clock running times of our improved algorithm in Figure 3. We exclude parsing and loading of queries and KBs as we aim to only evaluate our algorithm. Running times indicate an exponential dependency on the data size. There are also dependencies on the benchmark

type, e.g., for Φ_2 , where the non-existence of parking vehicles on the X-crossing improves performance, and Φ_4 , where more lanes on the X-crossing increases running time. This answers the first question positively, as our approach terminates in minutes to hours, with the lowest being 25.54 seconds for Φ_1 on the 20 second T-crossing scenario. However, the timeout was reached for Φ_4 on the X-crossing and $N \geq 2$ for reasons to be discussed later.

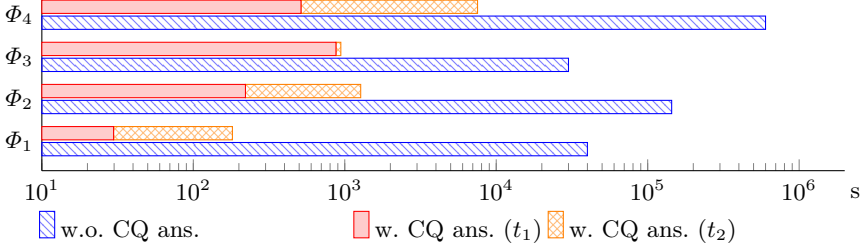


Fig. 4. Log-scaled running times with and without the CQ answering optimization enabled for the TOBM T-crossing $S = 0, N = 3$. Running times without the optimization are extrapolated after one hour.

The second question is addressed by comparing the running time of the improved algorithm to the basic algorithm from Algorithm 1. The results in Figure 4 show that the naïve approach fails for real-world data, even for two answer variables. Moreover, most of the time is still spent using the expensive, full semantics check despite iterating only through a fraction of all candidates (cf. Table 1). Hence, leveraging the CQ engine makes MTCQ answering practically feasible. However, some queries may trigger special cases in the optimizations of the CQ engine, leading to higher running times, e.g., role inclusion axioms for Φ_3 .

The strong effect of leveraging CQ answering motivates deeper examination. For this final question, we show wall clock times of both the CQ answering run t_1 ('first run') and the full-semantics run t_2 ('second run') in Figure 4. The effect of CQ answering can be twofold: Firstly, a set of candidates can be excluded globally. Secondly, even if a candidate was not globally excluded, it generates 'local' (non-)answers that can be cached for subsequent checks of Point 1 of Lemma 1. We thus report both exclusions, averaged over all time points and checked edges at each time point, in Table 1. Moreover, one can ask whether the second run is actually worthwhile. Table 1 reports how many certain answers ($\text{cert}_{\mathcal{K}}$) were already found in the first run ($\text{cert}_{\mathcal{K}}^1$).

Our results show CQ answering to aid mainly by excluding candidates globally in a highly-optimized fashion, as it can resort to techniques like binary instance retrieval, and often avoids consistency checks [36]. Local exclusion has minor but non-negligible effects, e.g., avoiding on average 42 additional candidates for Φ_3 . Moreover, all certain answers were already found in the first run, indicating suitability of using only the incomplete first run.

Table 1. Effects of CQ answering on MTCQ answering for the TOBM T-crossing $S = 0, N = 3$.

Query	Φ_1	Φ_2	Φ_3	Φ_4
Globally excluded candidates (%)	97.88	99.29	97.88	99.71
Globally and locally excluded candidates (%)	98.73	99.55	99.54	99.80
$ \text{cert}_{\mathcal{K}}^1 / \text{cert}_{\mathcal{K}} $	1	1	1	1

However, leveraging CQ answering has its limitations. For Φ_4 on the X-crossing and $N = 2$, the first run excluded 99.83% of all candidates after 2.38 minutes, leaving 960 candidates for the second run. However, this is no small task: for 200 time points in the data this leaves 180 seconds per time point to finish within 10 hours. Hence, each candidate must not take up more than 0.1875 seconds per time point on average, which entails checking multiple edges in multiple states. Experiments indicate each edge check to take a two-digit millisecond duration. Thus, to efficiently handle large candidate sets in the second run, we require further optimizations.

7 Conclusion

In this work, we introduced MTCQs as a suitable tool for situation recognition when testing requirements in complex operational domains, as illustrated by urban automated driving. Our tool, based on OPENLLET, brings MTCQ answering into practice by leveraging efficient CQ answering algorithms. Our custom benchmarks on safety-critical traffic situations show feasibility of our implementation for test evaluation settings and a potential to use our tool in other domains. These include risk assessments of other automated transportation systems, e.g., trams, maritime vessels, or delivery robots, and big-data analyses, e.g., process mining in business applications over intricate real-world structures.

As future work, we plan to investigate both practical optimizations and theoretical adaptations for increasing performance. For the former, it is interesting to (i) study how one can reuse query answers in consecutive time points given that potentially only small portions of the data change, (ii) identify fragments of MTCQs that can be answered more efficiently in practice (e.g., for runtime verification), and (iii) treat the spatial information more efficiently. On the theoretical side, it is interesting to study *rewriting* approaches, where the idea is to reduce the computation of certain answers to query evaluation in a target logic such as first-order logic (possibly with $+$, $<$) or DatalogMTL [39]. The benefit of such rewriting approaches is that one can leverage existing systems for evaluation in the target language. First-order rewritings have been studied in the context of more lightweight ontology and query languages [4]. While query rewritings need not exist in general (for complexity reasons), they might be very fruitful for practically occurring queries and ontologies.

References

1. Arechiga, N.: Specifying safety of autonomous vehicles in signal temporal logic. In: 2019 IEEE Intelligent Vehicles Symposium. pp. 58–63. IEEE, New York, USA (2019)
2. Artale, A., Franconi, E.: A survey of temporal extensions of description logics. *Annals of Mathematics and Artificial Intelligence* **30**(1), 171–210 (Jun 2000)
3. Artale, A., Kontchakov, R., Kovtunova, A., Ryzhikov, V., Wolter, F., Zakharyashev, M.: Ontology-mediated query answering over temporal data: A survey (invited talk). In: Schewe, S., Schneider, T., Wijsen, J. (eds.) 24th International Symposium on Temporal Representation and Reasoning, TIME 2017, October 16–18, 2017, Mons, Belgium. LIPIcs, vol. 90, pp. 1:1–1:37. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). <https://doi.org/10.4230/LIPIcs.TIME.2017.1>, <https://doi.org/10.4230/LIPIcs.TIME.2017.1>
4. Artale, A., Kontchakov, R., Kovtunova, A., Ryzhikov, V., Wolter, F., Zakharyashev, M.: First-order rewritability and complexity of two-dimensional temporal ontology-mediated queries. *Journal of Artificial Intelligence Research* **75**, 1223–1291 (2022). <https://doi.org/10.1613/jair.1.13511>, <https://doi.org/10.1613/jair.1.13511>
5. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyashev, M.: A cookbook for temporal conceptual data modelling with description logics. *ACM Transactions on Computational Logic* **15**(3), 25:1–25:50 (2014). <https://doi.org/10.1145/2629565>, <https://doi.org/10.1145/2629565>
6. Artale, A., Mazzullo, A., Ozaki, A.: Temporal description logics over finite traces. In: Ortiz, M., Schneider, T. (eds.) Proceedings of the 31st International Workshop on Description Logics co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), Tempe, Arizona, US, October. CEUR Workshop Proceedings, vol. 2211. CEUR-WS.org (2018)
7. ASAM e.V.: Openxontology user guide 1.0.0 (01 2022), <https://www.asam.net/standards/asam-openxontology/>, hoehenkirchen, Germany. Standard
8. Baader, F., Borgwardt, S., Lippmann, M.: Temporalizing ontology-based data access. In: Bonacina, M.P. (ed.) Automated Deduction – CADE-24. pp. 330–344. Springer, Berlin, Germany (2013)
9. Baader, F., Borgwardt, S., Lippmann, M.: Temporal query entailment in the description logic shq. *Journal of Web Semantics* **33**, 71–93 (2015). <https://doi.org/10.1016/j.websem.2014.11.008>
10. Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D.: The description logic handbook: Theory, implementation and applications. Cambridge University Press (2003)
11. Babisch, S., Neurohr, C., Westhofen, L., Schoenawa, S., Liers, H., et al.: Leveraging the gidas database for the criticality analysis of automated driving systems. *Journal of Advanced Transportation* **2023** (2023)
12. Borgwardt, S., Lippmann, M., Thost, V.: Temporal query answering in the description logic dl-lite. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) *Frontiers of Combining Systems*. pp. 165–180. Springer, Berlin, Germany (2013)
13. Borgwardt, S., Thost, V.: Temporal query answering in the description logic el. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. pp. 2819–2825. AAAI Press, Palo Alto, USA (2015)
14. Brandt, S., Kalaycı, E.G., Kontchakov, R., Ryzhikov, V., Xiao, G., Zakharyashev, M.: Ontology-based data access with a horn fragment of metric temporal logic. In:

- Proceedings of the AAAI Conference on Artificial Intelligence. vol. 31. AAAI Press, Palo Alto, USA (2017)
15. Chomicki, J.: Polynomial time query processing in temporal deductive databases. In: Rosenkrantz, D.J., Sagiv, Y. (eds.) *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. pp. 379–391. ACM Press, New York, USA (1990). <https://doi.org/10.1145/298514.298589>, <https://doi.org/10.1145/298514.298589>
16. De Gelder, E., Manders, J., Grappiolo, C., Paardekoooper, J.P., Den Camp, O.O., De Schutter, B.: Real-world scenario mining for the assessment of automated vehicles. In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. pp. 1–8. IEEE, New York, USA (2020)
17. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. pp. 854–860. AAAI Press, Palo Alto, USA (2013)
18. Elspas, P., Langner, J., Aydinbas, M., Bach, J., Sax, E.: Leveraging regular expressions for flexible scenario detection in recorded driving data. In: *2020 IEEE International Symposium on Systems Engineering (ISSE)*. pp. 1–8. IEEE, New York USA (2020)
19. Esterle, K., Gressenbuch, L., Knoll, A.: Formalizing traffic rules for machine interpretability. In: *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*. pp. 1–7. IEEE (2020)
20. Giacomo, G.D., Favorito, M.: Compositional approach to translate Itlf/ldlf into deterministic finite automata. In: Biundo, S., Do, M., Goldman, R., Katz, M., Yang, Q., Zhuo, H.H. (eds.) *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling*. pp. 122–130. AAAI Press, Palo Alto, USA (2021)
21. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics* **3**(2), 158–182 (2005)
22. Gutiérrez-Basulto, V., Jung, J.C., Kontchakov, R.: Temporalized EL ontologies for accessing temporal data: Complexity of atomic queries. In: Kambhampati, S. (ed.) *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. pp. 1102–1108. IJCAI/AAAI Press, Palo Alto, USA (2016)
23. Gutiérrez-Basulto, V., Jung, J.C., Ozaki, A.: On metric temporal description logics. In: Kaminka, G.A., Fox, M., Bouquet, P., Hüllermeier, E., Dignum, V., Dignum, F., van Harmelen, F. (eds.) *ECAI 2016 - 22nd European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications*, vol. 285, pp. 837–845. IOS Press, Amsterdam, The Netherlands (2016). <https://doi.org/10.3233/978-1-61499-672-9-837>
24. Horrocks, I., Kutz, O., Sattler, U.: The irresistible SRIQ. In: Grau, B.C., Horrocks, I., Parsia, B., Patel-Schneider, P.F. (eds.) *Proceedings of the OWLED*05 Workshop on OWL: Experiences and Directions*, Galway, Ireland, November 11–12, 2005. *CEUR Workshop Proceedings*, vol. 188. CEUR-WS.org (2005)
25. Horrocks, I., Tessaris, S.: A conjunctive query language for description logic aboxes. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. pp. 399–404. AAAI Press, Palo Alto, USA (2000)
26. Hülhnagen, T., Dengler, I., Tamke, A., Dang, T., Breuel, G.: Maneuver recognition using probabilistic finite-state machines and fuzzy logic. In: *2010 IEEE Intelligent Vehicles Symposium*. pp. 65–70. IEEE, New York, USA (2010)

27. Hummel, B.: Description logic for scene understanding at the example of urban road intersections. Ph.D. thesis, Universität Karlsruhe (TH) (2009)
28. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib. In: Kroening, D., Păsăreanu, C.S. (eds.) *Computer Aided Verification*. pp. 487–495. Springer, Berlin, Germany (2015)
29. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for mission-time ltl. In: Dillig, I., Tasiran, S. (eds.) *Computer Aided Verification*. pp. 3–22. Springer, Berlin, Germany (2019)
30. Lippmann, M.: Temporalised description logics for monitoring partially observable events. Ph.D. thesis, Dresden University of Technology (2014)
31. Lucchetti, A., Ongini, C., Formentin, S., Savaresi, S.M., Del Re, L.: Automatic recognition of driving scenarios for adas design. In: *IFAC Proceedings Volumes, Symposium on Advances in Automotive Control*. vol. 49, pp. 109–114. Elsevier, Amsterdam, The Netherlands (2016)
32. Lutz, C., Wolter, F., Zakharyashev, M.: Temporal description logics: A survey. In: *2008 15th International Symposium on Temporal Representation and Reasoning*. pp. 3–14. IEEE, New York, USA (2008)
33. Maierhofer, S., Rettinger, A.K., Mayer, E.C., Althoff, M.: Formalization of interstate traffic rules in temporal logic. In: *2020 IEEE Intelligent Vehicles Symposium (IV)*. pp. 752–759. IEEE, New York, USA (2020)
34. Neurohr, C., Westhofen, L., Henning, T., de Graaff, T., Möhlmann, E., Böde, E.: Fundamental Considerations around Scenario-Based Testing for Automated Driving. In: *2020 IEEE Intelligent Vehicles Symposium*. pp. 121–127. IEEE, New York, USA (2020). <https://doi.org/10.1109/IV47402.2020.9304823>
35. SAE International: J3016: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles (2021), Warrendale, USA. Standard
36. Sirin, E., Parsia, B.: Optimizations for answering conjunctive abox queries: First results. In: *Proc. of the 2006 Int. Workshop on Description Logics (DL'06)*. pp. 215–222 (2006)
37. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. *Journal of Web Semantics* **5**(2), 51–53 (2007)
38. Thost, V., Holste, J., Özçep, Ö.: On implementing temporal query answering in dl-lite. In: *Proc. of the 28th Int. Workshop on Description Logics (DL'15)*. vol. 1350, pp. 552–555 (2015)
39. Walega, P.A., Grau, B.C., Kaminski, M., Kostylev, E.V.: Datalogmtl over the integer timeline. In: Calvanese, D., Erdem, E., Thielscher, M. (eds.) *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*. pp. 768–777. IJCAI/AAAI Press, Palo Alto, USA (2020). <https://doi.org/10.24963/kr.2020/79>, <https://doi.org/10.24963/kr.2020/79>
40. Wang, D., Hu, P., Walega, P.A., Grau, B.C.: Meteor: practical reasoning in datalog with metric temporal operators. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. vol. 36, pp. 5906–5913. AAAI Press, Palo Alto, USA (2022)
41. Westhofen, L.: Openllet Temporal Query Benchmarks (Dec 2023). <https://doi.org/10.5281/zenodo.10436065>, <https://doi.org/10.5281/zenodo.10436065>
42. Westhofen, L., Neurohr, C., Butz, M., Scholtes, M., Schuldes, M.: Using ontologies for the formalization and recognition of criticality for automated driving. *IEEE Open Journal of Intelligent Transportation Systems* **3**, 519–538 (2022)

43. Westhofen, L., Stierand, I., Becker, J.S., Möhlmann, E., Hagemann, W.: Towards a congruent interpretation of traffic rules for automated driving-experiences and challenges. In: Proceedings of the International Workshop on Methodologies for Translating Legal Norms into Formal Representations (LN2FR 2022) in association with the 35th International Conference on Legal Knowledge and Information Systems (JURIX 2022). pp. 8–21 (2022)
44. Zipfl, M., Koch, N., Zöllner, J.M.: A comprehensive review on ontologies for scenario-based testing in the context of autonomous driving. In: 2023 IEEE Intelligent Vehicles Symposium. pp. 1–7. IEEE, New York, USA (2023). <https://doi.org/10.1109/IV55152.2023.10186681>

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 Boolean Separation Logic via Small Models *

Tomáš Dacík¹ , Adam Rogalewicz¹ , Tomáš Vojnar¹ , and Florian Zuleger²

¹ Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
idacik@fit.vut.cz

² Faculty of Informatics, Vienna University of Technology, Vienna, Austria

Abstract. We present a novel decision procedure for a fragment of separation logic (SL) with arbitrary nesting of separating conjunctions with boolean conjunctions, disjunctions, and guarded negations together with a support for the most common variants of linked lists. Our method is based on a model-based translation to SMT for which we introduce several optimisations—the most important of them is based on bounding the size of predicate instantiations within models of larger formulae, which leads to a much more efficient translation of SL formulae to SMT. Through a series of experiments, we show that, on the frequently used symbolic heap fragment, our decision procedure is competitive with other existing approaches, and it can outperform them outside the symbolic heap fragment. Moreover, our decision procedure can also handle some formulae for which no decision procedure has been implemented so far.

1 Introduction

In the last decade, separation logic (SL) [15, 30] has become one of the most popular formalisms for reasoning about programs working with dynamically-allocated memory, including approaches based on deductive verification [32], abstract interpretation [34], symbolic execution [31], or bi-abductive analysis [6, 12, 18]. The key ingredients of SL used in these approaches include the separating conjunction $*$, which allows modular reasoning by stating that the program heap can be decomposed into disjoint parts satisfying operands of the separating conjunction, along with inductive predicates describing shapes of data structures, such as lists, trees, or their various combinations.

The high expressive power of SL comes with the price of high complexity and even undecidability when several of its features are combined together. The existing decision procedures are usually limited to the so-called *symbolic heap fragment* that disallows any boolean structure of spatial assertions.

In this paper, we present a novel decision procedure for a fragment of SL that we call *boolean separation logic* (BSL). The fragment allows arbitrary nesting of separating conjunctions and boolean connectives of conjunction, disjunction, and a limited form of negation of the form $\varphi \wedge \neg\psi$ called *guarded negation*. To the best of our knowledge, no existing, practically applicable decision procedure supports a fragment with such a rich boolean structure and at least basic inductive predicates. The decision procedure for SL in CVC5 [29] supports arbitrary nesting of boolean connectives (including even unguarded negation, which is considered very expensive in the context of SL) but no inductive predicates. A support for conjunctions and disjunctions under separating

* The work was supported by the Czech Science Foundation project GA23-06506S. Basic research funding of the Czech team was provided by the FIT BUT internal project FIT-S-23-8151 and the ERC.CZ project LL1908. Tomáš Dacík was supported by the Brno Ph.D. Talent Scholarship funded by the Brno City Municipality.

conjunctions is available in the backend solver of the GRASSHOPPER verifier [27, 28] though not described in the papers. In our experimental evaluation, we outperform both of these approaches on some benchmarks (and can decide some formulae beyond the capabilities of both of them). We further show that adding guarded negations to BSL makes its satisfiability problem PSPACE-hard.

To motivate the usefulness of the fragment we consider, we now give several examples when SL formulae with a rich boolean structure are useful. First, in symbolic execution of heap manipulating programs, one usually needs to consider functions that involve some non-determinism—typically, at least the `malloc` statement has the non-deterministic contract $\{\text{emp}\} \ x = \text{malloc}() \ \{x \mapsto f \vee (x = \text{nil} \wedge \text{emp})\}$ (where f is a fresh variable) stating that when the statement is started in the empty heap, once it finishes, x is either allocated, or the allocation had failed and the heap is empty. Such contracts typically need a dedicated (and usually incomplete) treatment when no support of disjunctions is available.³ Further, the guarded negation semantically represents the set of counterexamples of the entailment $\varphi \models \psi$, and hence allows one to reduce entailment queries to UNSAT checking. Guarded negation can also be used when one needs to obtain several models of a formula φ by joining formulae representing the already obtained models to φ using guarded negations. One can also use the guarded negation to express interesting properties such as the fact that given a list $\text{sls}(x, y)$ and a pointer $y \mapsto z$, the pointer does not point back somewhere into the list closing a lasso. This can be expressed through the formula $(\text{sls}(x, y) \wedge \neg(\text{sls}(x, z) * \text{sls}(z, y))) * y \mapsto z$. Finally, boolean connectives can be introduced by translating quantitative separation logic into the classical SL [2].

In this work, we consider BSL with three fixed, built-in inductive predicates representing the most-common variants of lists: singly-linked (SLL), doubly-linked (DLL), and nested singly-linked (NLL). Our results can be easily extended for their variations such as nested doubly-linked lists of singly-linked lists and the like, but for the price of manually defining their semantics in the SMT encoding. We do, however, believe that our approach of bounding the sizes of models and instantiations of the individual predicates can be lifted to more complex inductive definitions and can serve as a starting point for allowing integration of SL with inductive definitions into SMT.

Contributions. Our approach to deciding BSL formulae is inspired by previous works on translation of SL to SMT. The early works [27] and [28] translate SL to intermediate theories first. Our approach is closer to the more recent approach of [16], which builds on small-model properties and axiomatizes reachability through pointer links directly. We extend the SL fragment considered in [16] by going beyond the so-called unique footprint property (under which it is much easier to obtain an efficient translation). Further, we define a more precise way to obtain global bounds on models of entire formulae, and, most importantly, we modify the translation of inductive predicates in a way that allows us to encode them succinctly by computing local bounds on their instantiations. According to our experiments, this makes the decision procedure efficient and competitive with the state-of-the-art approaches on the symbolic heap fragment (despite the increased decisive power). The claims we make in this paper are proven in [9].

³ Note that, while the post-condition with a single disjunction might seem simple, the formulae typically start growing in the further symbolic execution.

Related work. In [3], a proof system for deciding entailments of symbolic heaps with lists was proposed. This problem was later shown to be solvable in polynomial time in [8] via graph homomorphism checking. A superposition-based calculus for the fragment was presented in [23], and a model-based approach enhancing SMT solvers was proposed in [24]. In [24], a combination of SL with SMT theories is considered but still limited to the symbolic heap fragment. A more expressive boolean structure and integration with SMT theories was developed in [27] for lists and extended for trees in [28] but still without a support for guarded negations.

Other decision procedures are focusing on more general, *user-defined* inductive predicates (usually of some restricted form). They are based, e.g., on *cyclic proof systems* (CYCLIST [5], S2S [19, 20]); lemma synthesis (SONGBIRD [33]); or automata—tree automata are used in the tools SLIDE [13] and SPEN [11], and a specialised type of automata, called *heap automata*, is used in HARRSH [17]. These procedures do, however, not support nested use of boolean connectives and separating conjunctions.

There also exist works on deciding much more expressive fragments of SL such as [10, 14, 21, 26] but they do not lead to practically implementable decision procedures.

2 Preliminaries

Partial functions. We write $f : X \rightarrow Y$ to denote a *partial function* from X to Y . For a partial function f , $\text{dom}(f)$ and $\text{img}(f)$ denote its domain and image, respectively; $|f| = |\text{dom}(f)|$ denotes its size, and $f(x) = \perp$ denotes that f is undefined for x . A restriction $f|_A$ of f to $A \subseteq X$ is defined as $f(x)$ for $x \in A$ and undefined otherwise. To represent a finite partial function f , we often use the set notation $f = \{x_1 \mapsto y_1, \dots, x_n \mapsto y_n\}$ meaning that f maps each x_i to y_i , and is undefined for other values. We call partial functions f_1 and f_2 *disjoint* if $\text{dom}(f_1) \cap \text{dom}(f_2) = \emptyset$ and define their *disjoint union* $f_1 \uplus f_2$ as $f_1 \cup f_2$, which is otherwise undefined.

Graphs and paths. Let $G = (V, \rightarrow_1, \dots, \rightarrow_m)$ be a directed graph with vertices V and edges $\rightarrow = \rightarrow_1 \cup \dots \cup \rightarrow_m$. For $1 \leq f \leq m$, a sequence $\sigma = \langle v_0, v_1, \dots, v_n \rangle \in V^+$ is a path from v_0 to v_n via \rightarrow_f in G , denoted as $\sigma : v_0 \rightsquigarrow_f v_n$, if all elements of σ are distinct, and for all $0 \leq i < n$, it holds that $v_i \rightarrow_f v_{i+1}$. By the definition, paths cannot be cyclic. The *domain* of the path σ is the set $\text{dom}(\sigma) = \{v_0, v_1, \dots, v_{n-1}\}$, and the length of the path is defined as $|\sigma| = |\text{dom}(\sigma)| = n$.

Formulae. For a first-order formula φ , we denote by $\varphi[t/x]$ the formula obtained by simultaneously replacing all free occurrences of the variable x in φ with the term t . For a first-order model \mathcal{M} and a term t , we write $t^{\mathcal{M}}$ to denote the evaluation of t in \mathcal{M} defined as usual.

3 Separation Logic

Syntax. Let Vars be a countably infinite set of *sorted variables*. We denote by x^S a variable x of a sort $S \in \text{Sort} = \{\mathbb{S}, \mathbb{D}, \mathbb{N}\}$ representing a location in an SLL, DLL, or NLL, respectively. We omit the sorts when they are not relevant or clear from the context. We further assume that there exists a distinguished, unsorted variable nil . We write $\text{vars}(\varphi)$ to denote the set of all variables in φ plus nil (even when it does not appear in φ). Analogically, $\text{vars}_S(\varphi)$ stands for all variables of the sort S plus nil .

$(s, h) \models x \bowtie y$	iff $s(x) \bowtie s(y)$ and $\text{dom}(h) = \emptyset$ for $\bowtie \in \{=, \neq\}$
$(s, h) \models x \mapsto \langle f_i : f_i \rangle_{i \in I}$	iff $h = \{s(x) \mapsto \langle f_i : s(f_i) \rangle_{i \in I}\}$
$(s, h) \models \psi_1 \bowtie \psi_2$	iff $(s, h) \models \psi_1 \bowtie (s, h) \models \psi_2$ for $\bowtie \in \{\wedge, \vee, \neg\}$
$(s, h) \models \psi_1 * \psi_2$	iff $\exists h_1, h_2. h = h_1 \uplus h_2 \neq \perp$ and $(s, h_i) \models \psi_i$ for $i = 1, 2$
$(s, h) \models \exists x. \psi$	iff there exists ℓ such that $(s[x \mapsto \ell], h) \models \psi$
$(s, h) \models \text{sls}(x, y)$	iff $(s, h) \models x = y$, or $s(x) \neq s(y)$ and $(s, h) \models \exists n. x \mapsto n * \text{sls}(n, y)$
$(s, h) \models \text{dls}(x, y, x', y')$	iff $(s, h) \models x = y * x' = y'$, or $s(x) \neq s(y), s(x') \neq s(y')$, and $(s, h) \models \exists n. x \mapsto \langle n : n, p : y' \rangle * \text{dls}(n, y, x', x)$
$(s, h) \models \text{nls}(x, y, z)$	iff $(s, h) \models x = y$, or $s(x) \neq s(y)$ and $(s, h) \models \exists n, t. x \mapsto \langle n : n, t : t \rangle * \text{sls}(n, z) * \text{nls}(t, y, z)$

Fig. 1: The semantics of the separation logic. The existential quantifier is used for the definition of the semantics of inductive predicates and it is not a part of our fragment.

The syntax of our fragment is given by the following grammar:

$p ::= x^{\mathbb{S}} \mapsto \langle n : n \rangle \mid x^{\mathbb{D}} \mapsto \langle n : n, p : p \rangle \mid x^{\mathbb{N}} \mapsto \langle n : n, t : t \rangle$	(points-to predicates)
$\pi ::= \text{sls}(x^{\mathbb{S}}, y^{\mathbb{S}}) \mid \text{dls}(x^{\mathbb{D}}, y^{\mathbb{D}}, x_b^{\mathbb{D}}, y_b^{\mathbb{D}}) \mid \text{nls}(x^{\mathbb{N}}, y^{\mathbb{N}}, z^{\mathbb{S}})$	(inductive predicates)
$\varphi_A ::= x = y \mid x \neq y \mid p \mid \pi$	(atomic formulae)
$\varphi ::= \varphi_A \mid \varphi * \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \neg \varphi$	(formulae)

The *points-to* predicate $x \mapsto \langle f_1 : f_1, \dots, f_n : f_n \rangle$ denotes that x is a structure whose fields f_i point to values f_i . We often write $x \mapsto n$ instead of $x \mapsto \langle n : n \rangle$ and $x \mapsto _$ if the right-hand side is not relevant. We call x the *root* of the points-to predicate. If π is an inductive predicate $\text{sls}(x, y)$, $\text{dls}(x, y, x', y')$, or $\text{nls}(x, y, z)$, we again call x the root of π , y is the *sink* of π , and we write $\pi(x, y)$ to denote the root and the sink. We define the *sort* of the predicate π , denoted as S_π , as the sort of its root. Then, there is a one-to-one correspondence of predicates and sorts, which we often implicitly use.

Memory model. Let Loc be a countably infinite set of memory locations, and let $\text{Field} = \{n, p, t\}$ be the set of fields. A *stack* is a finite partial function $s : \text{Vars} \rightarrow \text{Loc}$. A *heap* is a finite partial function $h : \text{Loc} \rightarrow (\text{Field} \rightarrow \text{Loc})$. For succinctness, we write $h(\ell, f)$ instead of $h(\ell)(f)$. To represent heap elements in a readable way, we write functions $\text{Field} \rightarrow \text{Loc}$ as vectors with labels, i.e., $h(\ell) = \langle f : h(\ell, f) \mid f \in \text{Field} \wedge h(\ell, f) \neq \perp \rangle$ and we write $\text{img}(h)$ for $\{\ell \in \text{Loc} \mid \exists \ell', f. h(\ell', f) = \ell\}$. Moreover, we use $h(\ell) = n$ when $h(\ell) = \langle n : n \rangle$. A *stack-heap model* is a pair (s, h) where s is stack and h is a heap such that $s(\text{nil}) \neq \perp$ and $h(s(\text{nil})) = \perp$. We define the set of locations of the model (s, h) as $\text{locs}(s, h) = \text{img}(s) \cup \text{dom}(h) \cup \text{img}(h)$.

Semantics. The semantics of our SL over stack-heap models is given in Fig. 1. For pure formulae, we use the so-called *precise semantics*, which additionally requires that the heap must be empty⁴. The semantics of pointer assertions, boolean connectives, and

⁴ This is a common approach to avoid the atom true to be expressed as $\text{nil} = \text{nil}$. In our fragment, we forbid true in order not to introduce “unbounded” negations as $\neg \varphi \triangleq \text{true} \wedge \neg \varphi$. Due to this change, symbolic heaps are formulae of form $* \psi_i$ where each ψ_i is an atom.

separating conjunctions is as usual. The intuition behind the semantics of the inductive predicates is as follows. An SLL segment $\text{sls}(x, y)$ is either empty or represents an acyclic sequence of allocated locations starting from x and leading via the n field to y , which is not allocated. A DLL segment $\text{dls}(x, y, x', y')$ is either empty with $x = y$ and $x' = y'$, or it represents an acyclic sequence that is doubly-linked via the n and p fields and leads from the first allocated location x of the segment to its last allocated location x' (x and x' may coincide) with y/y' being the n/p -successors of x'/x , respectively. Both y and y' are not allocated. An NLL segment $\text{nls}(x, y, z)$ is a (possibly empty) acyclic sequence of locations starting from x and leading to y via the t (top) field in which successor of each locations starts a disjoint inner SLL to z via n .

Stack-heap graphs. We frequently identify stack-heap models with their graph representation. A stack-heap model (s, h) defines a graph $G[(s, h)] = (V, (\rightarrow_f)_{f \in \text{Field}})$ where $V = \text{locs}(s, h)$ and $u \rightarrow_f v$ iff $h(u, f) = v$. We frequently use the fact that if there exists a path $\sigma : x \rightsquigarrow_f y$ in a stack-heap graph, then it is uniquely determined because f -edges are given by a partial function.

4 Small-Model Property

Small-model properties, which state that each satisfiable formula has a model of bounded size, are frequently used for various fragments of SL to prove their decidability [7] or to design decision procedures [16, 26, 29]. The latter is also the case of our translation-based decision procedure which will heavily rely on enumeration over all locations, and, for its efficiency, it is therefore necessary to obtain location bounds that are as small as possible.

The way we obtain our small-model property is inspired by the approach of [16] and by insights from the so-called *strong-separation logic* [26]. The main idea is to define a satisfiability-preserving reduction $\downarrow^s h$ which takes a heap h (referenced from a stack s), decomposes it into basic sub-heaps (which we call *chunks*), and reduces it per the sub-heaps in such a way that its size can be easily bounded by a linear expression. To define the reduction, we first need to introduce some auxiliary notions related to stack-heap models.

We say that a model (s, h) is *positive* if there exists φ with $(s, h) \models \varphi$. A positive model (s, h) is *atomic* if it is non-empty, and for all positive models (s, h_1) and (s, h_2) , $h = h_1 \uplus h_2$ implies that $h_1 = \emptyset$ or $h_2 = \emptyset$. In other words, atomic models cannot be decomposed into two non-empty positive models. Several examples of atomic models are shown in Fig. 2. Observe that the models of dls (Figure 2b) and nls (Figure 2c) are indeed atomic as any of their decomposition, in particular the split at the location u , does not give two positive models.

A sub-heap $c \subseteq h$ is a *chunk* of a model (s, h) if c is a maximal sub-heap of h such that (s, c) is an atomic positive model. Notice that the way the definition of chunks is constructed excludes the possibility of using as a chunk a sub-heap of a heap that itself forms an atomic model. The reason is that otherwise the remaining part of the larger atomic model could not be described by the available predicates. For example, in nested lists as shown in Fig. 2c, one cannot take as a chunk a part of some inner list (e.g., the pointer $u \mapsto z$) as the heap shown in the figure itself forms an atomic model. Indeed, if $u \mapsto z$ was removed, one would need a more general version of the NLL predicate to cover the remaining heap by atomic models.

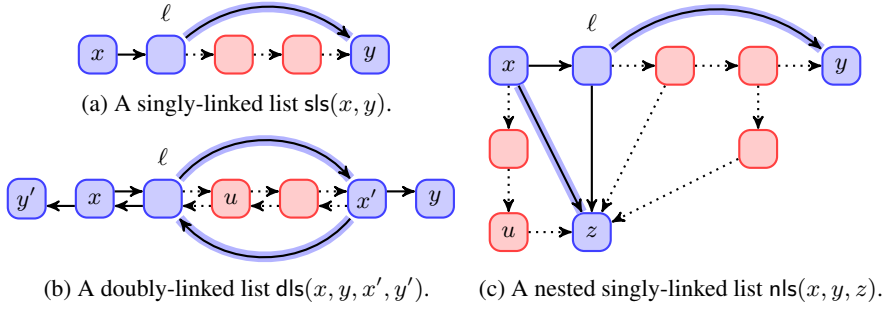


Fig. 2: An illustration of reductions of atomic models of inductive predicates. Removed heap locations are red, removed edges are dotted, and added edges are highlighted.

Lemma 1 (Chunk decomposition). *A positive model (s, h) can be uniquely decomposed into the set of its chunks, denoted $\text{chunks}(s, h)$, i.e., $h = \biguplus \text{chunks}(s, h)$.*

Minimal atomic models of inductive predicates. The key reason why the small-model property that we are going to state holds is that our fragment of SL cannot distinguish atomic models of the considered predicates beyond certain small sizes—namely, two for sls and nls, and three for dls. For further use, we will now state predicates describing exactly the sets of the indistinguishable lists of the different kinds.

We start with SLLs and use a disequality to exclude empty lists: $\text{sls}_{\geq 1}(x, y) \triangleq \text{sls}(x, y) * x \neq y$, and a guarded negation to exclude lists of length one consisting of a single pointer only: $\text{sls}_{\geq 2}(x, y) \triangleq \text{sls}_{\geq 1}(x, y) \wedge \neg(x \mapsto y)$. A similar predicate can be defined for NLLs too: $\text{nls}_{\geq 2}(x, y, z) \triangleq (\text{nls}(x, y, z) * x \neq y) \wedge \neg(x \mapsto \langle n: z, t: y \rangle)$.

For DLLs, we define $\text{dls}_{\geq 2}(x, y, x', y') \triangleq \text{dls}(x, y, x', y') * x \neq y * x' \neq x'$ to exclude models that are either empty or consist of a single pointer; and $\text{dls}_{\geq 3}(x, y, x', y') \triangleq \text{dls}_{\geq 2}(x, y, x', y') \wedge \neg(x \mapsto \langle n: x', p: y' \rangle * x' \mapsto \langle n: y, p: x \rangle)$ to also exclude models consisting of exactly two pointers.

It holds that atomic models, and consequently also chunks, are precisely either models of single pointers or of the above predicates.

Lemma 2. *For atomic model (s, h) , exactly one of the following conditions holds.*

1. $(s, h) \models x \mapsto _$ for some x . (pointer-atom)
2. $(s, h) \models \text{sls}_{\geq 2}(x, y)$ for some x and y . (sls-atom)
3. $(s, h) \models \text{dls}_{\geq 3}(x, y, x', y')$ for some x, y, x' , and y' . (dls-atom)
4. $(s, h) \models \text{nls}_{\geq 2}(x, y, z)$ for some x, y , and z . (nls-atom)

We can now define the reduction in the way we have already sketched.

Definition 1. *The heap of a positive model (s, h) reduces to $\downarrow^s h = \biguplus_{c \in \text{chunks}(s, h)} \downarrow^s c$ where the reduction of a chunk c with a root x as follows:*

- $\downarrow^s c = c$ if $(s, c) \models x \mapsto _$.
- $\downarrow^s c = \{s(x) \mapsto \ell, \ell \mapsto s(y)\}$ where $\ell = c(s(x), n)$ if $(s, c) \models \text{sls}_{\geq 2}(x, y)$ for some y .
- $\downarrow^s c = \{s(x) \mapsto \langle n: \ell, p: s(y') \rangle, \ell \mapsto \langle n: s(x'), p: s(x) \rangle, s(x') \mapsto \langle n: s(y), p: \ell \rangle\}$ where $\ell = c(s(x), n)$ if $(s, c) \models \text{dls}_{\geq 3}(x, y, x', y')$ for some x', y' and y .
- $\downarrow^s c = \{s(x) \mapsto \langle t: \ell, n: s(z) \rangle, \ell \mapsto \langle t: s(y), n: s(z) \rangle\}$ where $\ell = c(s(x), t)$ if $(s, c) \models \text{nls}_{\geq 2}(x, y, z)$ for some y and z .

We lift the reduction to stack-heap models as $\downarrow^X(s, h) = (s', \downarrow^{s'} h)$ where $s' = s|_X$ for some set of variables X and show that it preserves satisfiability when $X = \text{vars}(\varphi)$.

Theorem 1. *For a positive model (s, h) , it holds that $(s, h) \models \varphi$ iff $\downarrow^{\text{vars}(\varphi)}(s, h) \models \varphi$.*

The final step to show our small-model property is to find an upper bound on the size of the reduced models. We define the size of a variable x^S , $\|x^S\|$, which represents its contribution to the location bound, and is defined as 2 if $S \in \{\mathbb{S}, \mathbb{N}\}$ and 1.5 if $S = \mathbb{D}$ (this corresponds to the size of a reduced chunk of sort S divided by the number of variables which are allocated in it). We further define $\|\text{nil}\| = 0$. The location bound of φ is then given as $\text{bound}(\varphi) = 1 + \lfloor \sum_{x \in \text{vars}(\varphi)} \|x\| \rfloor$ (the additional location is for nil). Analogically, the location bound for a sort S is $\text{bound}_S(\varphi) = \lfloor \sum_{x \in \text{vars}_S(\varphi)} \|x\| \rfloor$.

Theorem 2 (Small-model property). *If a formula φ is satisfiable, then there exists a model $(s, h) \models \varphi$ such that $|\text{locs}(s, h)| \leq \text{bound}(\varphi)$.*

We conjecture that the bound can be further improved, e.g., by showing that each model can be transformed to an equivalent one (indistinguishable by BSL formulae) such that the number of its chunks is bounded by the number of roots of spatial predicates in φ . We demonstrate this on the formula $\text{sls}(x, y) * y \mapsto z$ and its model in which y points back into the middle of the list segment (thus splitting it into two chunks). Clearly, this model can be transformed by redirecting z outside of the list domain.

5 Translation-Based Decision Procedure

In this section, we present our translation of SL to SMT. We first present an SMT encoding of our memory model and a translation of basic predicates and boolean connectives. Then we discuss methods for efficient translation of separating conjunctions and inductive predicates with the focus on avoiding quantifiers by replacing them by small enumerations of their instantiations.

We fix an input formula φ and let $n_S = \text{bound}_S(\varphi)$ for each sort $S \in \text{Sort}$.

5.1 Encoding the Memory Model in SMT

To encode the heap, we use a classical approach which encodes its mapping and domain separately [16, 27, 29]. Namely, we use arrays to encode mappings and sets to encode domains. We also use the theory of datatypes to represent a finite sort of locations by a datatype $L \triangleq \text{loc}^{\text{nil}} \mid \text{loc}_1^{\mathbb{S}} \mid \dots \mid \text{loc}_{n_S}^{\mathbb{S}} \mid \text{loc}_1^{\mathbb{D}} \mid \dots \mid \text{loc}_{n_D}^{\mathbb{D}} \mid \text{loc}_1^{\mathbb{N}} \mid \dots \mid \text{loc}_{n_N}^{\mathbb{N}}$.

Now, we define the signature of the translation's language over the sort L . For each $x \in \text{vars}(\varphi)$, we introduce a constant x of the same name—its interpretation represents the stack image $s(x)$. To represent the heap, we introduce a set symbol D representing the domain and an array symbol h_f for each field $f \in \text{Field}$ which represents the mapping of the partial function $\lambda \ell. h(\ell, f)$. To distinguish sorts of locations, we further introduce a set symbol D_S for each sort $S \in \text{Sort}$. We define meaning of these symbols by showing how a stack-heap model can be reconstructed from a first-order model.

Definition 2 (Inverse translation). *Let \mathcal{M} be a first-order model. We define its inverse translation $T_\varphi^{-1}(\mathcal{M}) = (s, h)$ where $s(x) = x^{\mathcal{M}}$ if $x \in \text{vars}(\varphi)$ and*

$$h(\ell) = \begin{cases} \langle n: h_n[\ell]^{\mathcal{M}} \rangle & \text{if } \ell \in (D \cap D_{\mathbb{S}})^{\mathcal{M}} \\ \langle n: h_n[\ell]^{\mathcal{M}}, p: h_p[\ell]^{\mathcal{M}} \rangle & \text{if } \ell \in (D \cap D_{\mathbb{D}})^{\mathcal{M}} \\ \langle n: h_n[\ell]^{\mathcal{M}}, t: h_t[\ell]^{\mathcal{M}} \rangle & \text{if } \ell \in (D \cap D_{\mathbb{N}})^{\mathcal{M}}. \end{cases}$$

To ensure consistency of the translation with the memory model used, we define the following axioms that a result of translation needs to satisfy:

$$\mathcal{A}_\varphi \triangleq \text{nil} = \text{loc}^{\text{nil}} \wedge \text{nil} \notin D \wedge \bigwedge_{S \in \text{Sort}} (D_S = \{\text{loc}^{\text{nil}}, \text{loc}_1^S, \dots, \text{loc}_{n_S}^S\} \wedge \bigwedge_{x \in \text{vars}_S(\varphi)} x \in D_S).$$

The axioms ensure that nil is never allocated, that each variable is interpreted as a location of the corresponding sort and they fix the interpretation of the sets $D_{\mathbb{S}}, D_{\mathbb{D}}, D_{\mathbb{N}}$, which we will later use in the translation to assign sorts to locations.

5.2 Translation of SL to SMT

We define the translation as a function $T(\varphi) = \mathcal{A}_\varphi \wedge T(\varphi, D)$ where \mathcal{A}_φ are the above defined axioms and $T(\varphi, D)$ is a recursive translation function of the formula φ with the domain symbol D . The translation $T(\cdot)$ together with the inverse translation of models $T_\varphi^{-1}(\cdot)$ are linked by the following correctness theorem.

Theorem 3 (Translation correctness). *An SL formula φ is satisfiable iff its translation $T(\varphi)$ is satisfiable. Moreover, if $\mathcal{M} \models T(\varphi)$, then $T_\varphi^{-1}(\mathcal{M}) \models \varphi$.*

The translation of non-inductive predicates and boolean connectives is defined as:

$$\begin{aligned} T(x \bowtie y, F) &\triangleq x \bowtie y \wedge F = \emptyset && \text{for } \bowtie \in \{=, \neq\} \\ T(\psi_1 \bowtie \psi_2, F) &\triangleq T(\psi_1, F) \bowtie T(\psi_2, F) && \text{for } \bowtie \in \{\wedge, \vee, \wedge \neg\} \\ T(x \mapsto \langle f_i : f_i \rangle_{i \in I}, F) &\triangleq F = \{x\} \wedge \bigwedge_{i \in I} h_{f_i}[x] = f_i \end{aligned}$$

The translation of boolean connectives follows the boolean structure and propagates the domain symbol F to the operands. The translation of pointer assertions postulates content of memory cells represented by arrays and also requires the domain F to be $\{x\}$.

Translation of separating conjunctions. The semantics of separating conjunctions involves a quantification over sets (heap domains). The most direct way of translation is to use quantifiers over sets leading to decidable formulae due to the bounded location domain. This approach combined with a counterexample-guided quantifier instantiation is used in the decision procedure for a fragment of SL supported in CVC5 [29]. In some fragments, however, separating conjunctions can be translated in a way that completely avoids quantifiers. An example is the fragment of boolean combinations of symbolic heaps which has the so-called *unique footprint property* (UFP) [16, 27]—a formula ψ has a (unique) footprint in a model (s, h) with $(s, h) \models \psi * \text{true}$ ⁵, if there exists a (unique) set F such that $(s, h|_F) \models \psi$. The UFP-based approaches of [16, 27] axiomatize the footprints during translation and check operands of separating conjunctions just on the sub-heaps induced by their footprints.

However, UFP does not hold for BSL because of disjunctions. As an example, take the formula $\psi \triangleq x \mapsto y \vee \text{emp}$ and the heap $h = \{x \mapsto y\}$. Both $(s, h|_{\{s(x)\}}) \models \psi$ and $(s, h|_{\emptyset}) \models \psi$ hold. The sets $\{s(x)\}$ and \emptyset are, however, the only footprints of ψ in (s, h) , and this observation can be used to generalise the idea of footprints beyond the fragment in which they are unique.

⁵ Assuming the standard semantics of true which is not part of our logic.

Instead of axiomatizing the footprints, our translation builds a set of footprint terms for operands of separating conjunctions. This change can be also seen as a simplification of the former translations as it eliminates the need to deal with two kinds of formulae (the actual translation and footprint axioms), which must be treated differently during the translation. However, the precise computation of the set of all footprints of ψ in (s, h) , denoted as $\text{FP}_{(s,h)}(\psi)$, is as hard as satisfiability—when the set of footprints is non-empty, the formula ψ is satisfiable. Therefore, we compute just an over-approximation denoted as $\text{FP}^\#(\psi)$. This is justified by the following lemma which gives an equivalent semantics of the separating conjunction in terms of footprints.

Lemma 3. *Let $\varphi \triangleq \psi_1 * \psi_2$ and let (s, h) be a model. Let \mathcal{F}_1 and \mathcal{F}_2 be sets of locations such that $\text{FP}_{(s,h)}(\psi_i) \subseteq \mathcal{F}_i$. Then $(s, h) \models \psi_1 * \psi_2$ iff*

$$\bigvee_{F_1 \in \mathcal{F}_1} \bigvee_{F_2 \in \mathcal{F}_2} \bigwedge_{i=1,2} (s, h|_{F_i}) \models \psi_i \wedge F_1 \cap F_2 = \emptyset \wedge F_1 \cup F_2 = \text{dom}(h).$$

Intuitively, to check whether a separating conjunction holds in a model, it is not necessary to check all possible splits of the heap, but only the splits induced by (possibly over-approximated) footprints of its operands. The lemma is therefore a generalisation of UFP and leads to the following definition of the translation $\mathsf{T}(\psi_1 * \psi_2, F)$:

$$\exists F_1 \in \mathcal{F}_1. \exists F_2 \in \mathcal{F}_2. \mathsf{T}(\psi_1, F_1) \wedge \mathsf{T}(\psi_2, F_2) \wedge F_1 \cap F_2 = \emptyset \wedge F = F_1 \cup F_2.$$

Here, we use a quantifier expression of the form $\exists x \in X. \psi$ as a placeholder that helps us to define two methods which the translation can use for separating conjunctions:

- The method `SatEnum` computes sets of footprints \mathcal{F}_i as $\text{FP}^\#(\psi_i)$ (the computation is described below) and replaces expressions $\exists x \in X. \psi$ with $\bigvee_{x' \in X} \psi[x'/x]$ as in Lemma 3. This strategy is quite efficient in many practical cases when we can compute small sets of footprints \mathcal{F}_1 and \mathcal{F}_2 .
- The method `SatQuantif` does not compute sets \mathcal{F}_i at all and replaces $\exists x \in X. \psi$ simply with $\exists x. \psi$. This strategy is better when the existential quantifier can be later eliminated by Skolemization or when the set of footprints would be too large.

We now show how to compute the set of footprint terms $\text{FP}^\#(\psi)$. We again postpone inductive predicates to Section 5.3. We just note that their footprints are unique. The cases of pure formulae and pointer assertions follow directly from the definition of their semantics, which requires the heap to be empty and a single pointer, respectively.

$$\text{FP}^\#(x \bowtie y) = \{\emptyset\} \text{ for } \bowtie \in \{=, \neq\} \quad \text{FP}^\#(x \mapsto _) = \{\{x\}\}$$

For the boolean conjunction, we can select from footprints of its operand the one with the lesser cardinality. Since negations have many footprints (consider, e.g., $\neg \text{emp}$), we define the case of the guarded negation by taking footprints of its guard. The disjunction is the only case which brings non-uniqueness as we need to consider footprints of both of its operands.

$$\begin{aligned} \text{FP}^\#(\psi_1 \wedge \neg \psi_2) &= \text{FP}^\#(\psi_1) & \text{FP}^\#(\psi_1 \vee \psi_2) &= \text{FP}^\#(\psi_1) \cup \text{FP}^\#(\psi_2) \\ \text{FP}^\#(\psi_1 \wedge \psi_2) &= \text{if } |\text{FP}^\#(\psi_1)| \leq |\text{FP}^\#(\psi_2)| \text{ then } \text{FP}^\#(\psi_1) \text{ else } \text{FP}^\#(\psi_2) \end{aligned}$$

Finally, we define footprints of the separating conjunction by taking the union $F_1 \cup F_2$ for each pair (F_1, F_2) of footprints of its operands. Notice that here $F_1 \cup F_2$ represents an SMT term, therefore we cannot replace it with a disjoint union which is not available in the classical set theories in SMT. We can, however, use heuristics and filter out terms for which we can statically determine that interpretations of F_1 and F_2 are not disjoint.

$$\text{FP}^\#(\psi_1 * \psi_2) = \{F_1 \cup F_2 \mid F_1 \in \text{FP}^\#(\psi_1) \text{ and } F_2 \in \text{FP}^\#(\psi_2)\}$$

We state the correctness of the footprint computation in the following lemma.

Lemma 4. *Let \mathcal{M} be a first-order model with $\mathcal{M} \models \top(\varphi)$ and let $(s, h) = \top_\varphi^{-1}(\mathcal{M})$. Then we have $\text{FP}_{(s,h)}(\varphi) \subseteq \{F^\mathcal{M} \mid F \in \text{FP}^\#(\varphi)\}$.*

5.3 Translation of Inductive Predicates

To translate inductive predicates, we express them in terms of reachability and paths in the heaps. While unbounded reachability cannot be expressed in first-order logic, we can efficiently express bounded *linear* reachability in our encoding. The linearity means that each path uses only a single field (which is not the case, e.g., for paths in trees). All predicates in this section are parametrised with an interval $[m, n]$ which bounds the length of the considered paths. When we do not state the bounds explicitly, we assume conservative bounds $[0, \text{bound}_S(\varphi)]$ for a path starting from a root of a sort S . We show how to compute more precise bounds in Section 6. We start with the translation of reachability:

$$\text{reach}^{\overline{n}}(h, x, y) \triangleq h^n[x] = y \quad \text{reach}^{[m,n]}(h, x, y) \triangleq \bigvee_{m \leq i \leq n} \text{reach}^{\overline{i}}(h, x, y)$$

Here, the predicate $\text{reach}^{\overline{n}}(h, x, y)$ expresses that x can reach y via a field represented by the array h in exactly n steps. Similarly, $\text{reach}^{[m,n]}$ expresses reachability in m to n steps. Besides reachability, we will need a macro $\text{path}_C(h, x, y)$ expressing the domain of a path from x to y , or the empty set if such a path does not exist:

$$\begin{aligned} \text{path}_C^{\overline{n}}(h, x, y) &\triangleq \bigcup_{0 \leq i < n} C(h^i[x]) \\ \text{path}_C^{[m,n]}(h, x, y) &\triangleq \text{if } (\text{reach}^{\overline{m}}(h, x, y)) \text{ then } (\text{path}_C^{\overline{m}}(h, x, y)) \\ &\quad \dots \text{ else if } (\text{reach}^{\overline{n}}(h, x, y)) \text{ then } (\text{path}_C^{\overline{n}}(h, x, y)) \text{ else } (\emptyset) \end{aligned}$$

The additional parameter C is a function applied to each element of the path that can be used to define nested paths. We define a simple path $\text{path}_S^{[m,n]}(h, x, y) \triangleq \text{path}_C^{[m,n]}(h, x, y)$ with $C \triangleq \lambda \ell. \{\ell\}$ and a nested path as $\text{path}_N^{[m,n]}(h_1, h_2, x, y, z) \triangleq \text{path}_C^{[m,n]}(h_1, x, y)$ with $C \triangleq \lambda \ell. \text{path}_S(h_2, \ell, z)$. In the case of the nested path, the array h_1 represents the top-level path from x to y , and h_2 represents nested paths terminating in the common location z . Now we can define footprints of inductive predicates using path terms as follows:

$$\begin{aligned} \text{FP}^\#(\pi(x, y)) &= \{\text{path}_S(h_n, x, y)\} & \text{for } \pi \in \{\text{sls}, \text{dls}\} \\ \text{FP}^\#(\text{nls}(x, y, z)) &= \{\text{path}_N(h_t, h_n, x, y, z)\} \end{aligned}$$

The common part of the translation $T(\pi(x, y), F)$ postulates the existence of a top-level path from x to y and a domain F based on this path (formalised in the formula `main_path` below); and ensures that all locations have the correct sort (through the formula `typing`). For DLLs, we add an invariant which ensures that its locations are correctly doubly-linked (the `back_links` formula), and we further need a special treatment of the cases when the list is empty as well as a special treatment for its roots and sinks (cf. the formula `boundaries`). For NLLs, we add an invariant stating that an inner list starts from each location in its top-level path (the `inner_lists` formula) and that those inner paths are disjoint (the `disjoint` formula)⁶.

- $T(\text{sls}(x, y), F) \triangleq \text{main_path} \wedge \text{typing}$ where

$$\text{main_path} \triangleq \text{reach}(h_n, x, y) \wedge F = \text{path}_S(h_n, x, y) \text{ and } \text{typing} \triangleq F \subseteq D_S.$$
- $T(\text{dls}(x, y, x', y'), F) \triangleq \text{empty} \vee \text{nonempty}$ where

$$\begin{aligned} \text{empty} &\triangleq x = y \wedge x' = y' \wedge F = \emptyset, \\ \text{nonempty} &\triangleq x \neq y \wedge x' \neq y' \wedge \text{main_path} \wedge \text{boundaries} \wedge \text{typing} \wedge \text{back_links}, \\ \text{main_path} &\triangleq \text{reach}(h_n, x, y) \wedge F = \text{path}_S(h_n, x, y), \\ \text{boundaries} &\triangleq h_p[x] = y' \wedge h_n[x'] = y \wedge x' \in F \wedge y' \notin F, \\ \text{typing} &\triangleq F \subseteq D_{\mathbb{D}}, \\ \text{back_links} &\triangleq \forall \ell. (\ell \in F \wedge \ell \neq x') \rightarrow h_p[h_n[\ell]] = \ell. \end{aligned}$$
- $T(\text{nls}(x, y, z), F) \triangleq \text{main_path} \wedge \text{typing} \wedge \text{inner_lists} \wedge \text{disjoint}$ where

$$\begin{aligned} \text{main_path} &\triangleq \text{reach}(h_t, x, y) \wedge F = \text{path}_N(h_t, h_n, x, y, z), \\ \text{typing} &\triangleq \text{path}_S(h_t, x, y) \subseteq D_{\mathbb{N}} \wedge F \setminus \text{path}_S(h_t, x, y) \subseteq D_S, \\ \text{inner_lists} &\triangleq \forall \ell. \ell \in F \cap D_{\mathbb{N}} \rightarrow \text{reach}(h_n, h[\ell], z), \\ \text{disjoint} &\triangleq \forall \ell_1, \ell_2. (\{\ell_1, \ell_2\} \subseteq F \wedge \ell_1 \neq \ell_2 \wedge h_n[\ell_1] = h_n[\ell_2]) \rightarrow h_n[\ell_1] \notin F. \end{aligned}$$

Path quantifiers. Invariants of paths are naturally expressed using universal quantifiers. For quantifiers, however, we cannot directly take advantage of bounds on path lengths. Therefore, similarly as for separating conjunctions, we use the idea of replacing quantifiers by small enumerations of their instances, which is efficient when we can compute small enough bounds on the paths. For example, if we know that the length of an f -path with a root x is at most two, it is enough to instantiate its invariant for x , $h_f[x]$, and $h_f^2[x]$. This idea is formalised using expressions $\mathbb{P}_{(h,x)}^{\leq n} \ell. \psi$, which we call *path quantifiers* and which state that ψ holds for all locations of the path with the length n starting from x via the array h :

$$\mathbb{P}_{(h,x)}^{\leq n} \ell. \psi \triangleq \bigwedge_{0 \leq i \leq n} \psi[h^i[x]/\ell].$$

If we need to quantify over nested paths, we need to use two path quantifiers (one for the top-level path and one for the nested paths). The quantifiers in the last conjunct of the NLL translation can be rewritten as $\mathbb{P}_{(h_t,x)} \ell'_1. \mathbb{P}_{(h_t,x)} \ell'_2. \mathbb{P}_{(h_n,\ell'_1)} \ell_1. \mathbb{P}_{(h_n,\ell'_2)} \ell_2$. In this expression, ℓ'_1 and ℓ'_2 range over locations in the top-level list, and ℓ_1 and ℓ_2 range over locations in the nested paths starting from ℓ'_1 and ℓ'_2 , respectively.

⁶ In the consequent of the disjoint formula, we could also write $h_n[\ell_1] = z$ instead of $h_n[\ell_1] \notin F$, but the latter leads to better performance of SMT solvers.

5.4 Complexity

This section briefly discusses the complexity of the proposed decision procedure as well as the complexity lower bound for the satisfiability problem in the considered fragment of SL. We will use $\text{SAT}(\omega_1, \dots, \omega_n)$ to denote the satisfiability problem for a sub-fragment constructed of atomic formulae and the connectives ω_i and $\text{SAT}(\overline{\omega_1, \dots, \omega_n})$ to denote the fragment where none of the connectives ω_i appear.

Theorem 4. *The procedure SatQuantif produces formula of polynomial size, and, for $\text{SAT}(\overline{\wedge, \neg})$, it runs in NP. The procedure SatEnum runs in NP for $\text{SAT}(\overline{\vee})$.*

Proof (sketch). When not considering the instantiation of quantifiers over footprints, both SatQuantif and SatEnum produce a formula $T(\varphi)$ of a polynomial size dominated by the translation of inductive predicates. For the variant of the translation of inductive predicates using universal quantifiers over locations, the size is $\mathcal{O}(n^3)$ for SLLs and DLLs (dominated by the $\mathcal{O}(n^3)$ size of the path_S term), and $\mathcal{O}(n^5)$ for NLLs (dominated by path_N). If the input formula does not contain guarded negations, then all quantifiers can be eliminated using Skolemization. The translated formulae are then in a theory decidable in NP (e.g., when sets are encoded as extended arrays [22]).

The procedure SatEnum can produce exponentially large formulae because of the footprint enumeration. This can be prevented if the input formula does not contain disjunctions, in which case the footprints of all sub-formulae are unique, i.e., singleton sets. The translated formulae are then again in a theory decidable in NP. \square

Theorem 5. $\text{SAT}(\mapsto, \wedge, \neg, \wedge, \vee, *)$ is PSPACE-complete.

Proof (sketch). Membership in PSPACE was proved in [26] for a more expressive fragment. For the hardness part, we build on the reduction from QBF used in [7]. In this reduction, the boolean value of a variable is represented by the corresponding SL variable being allocated (always pointing to nil for simplicity). The fact that x is false is expressed using a negative points-to predicate stating that x is not allocated. The existential quantifier is expressed using the separating conjunction, and the universal quantifier is obtained using the (unguarded) negation. (For details, see [7].)

We show that this reduction can be done without the unguarded negation and the negative points-to assertion, using the guarded negation instead. The key observation is that, for a QBF formula with variables X , we can express that all variables in X can have arbitrary boolean values as $\text{arbitrary}[X] \triangleq *_{x \in X} (x \mapsto \text{nil} \vee \text{emp})$. In the context of variables X , we can then express negation as $\neg F \triangleq \text{arbitrary}[X] \wedge \neg F$ and the truth values of a variable x as $\neg x \triangleq \text{arbitrary}[X \setminus \{x\}]$ and $x \triangleq \text{arbitrary}[X] * x \mapsto \text{nil}$. The rest of the reduction then easily follows [7]. \square

6 Optimised Bound Computation

In many practical cases, the main source of complexity is the translation of inductive predicates, which heavily depends on the possible lengths of paths between locations. We now propose how to bound the length of these paths based on the so-called *SL-graphs* which are graph representations of constraints imposed by SL formulae. SL-graphs were originally used for representation and deciding of symbolic heaps with lists in [8]. Here, we use their generalised form which captures must-relations holding in all models of a given formula. Note that the nodes of the graphs are implicitly given by the domains of the involved relations, which themselves can be viewed as edges.

Definition 3. An SL-graph of φ is a tuple $G[\varphi] = (\ominus, \oplus, (\ominus_f, \ominus_{\bar{f}}, \odot_f)_{f \in \text{Field}})$ where:

- $\ominus \subseteq \text{vars}(\varphi) \times \text{vars}(\varphi)$ is an equivalence relation called *must-equality*,
- $\oplus \subseteq \text{vars}(\varphi) \times \text{vars}(\varphi)$ is a symmetric relation called *must-disequality*,
- $\ominus_f \subseteq \text{vars}(\varphi) \times \text{vars}(\varphi)$ is a *must-f-pointer* relation,
- $\ominus_{\bar{f}} \subseteq \text{vars}(\varphi) \times \text{vars}(\varphi)$ is an *irreflexive must-f-path* relation,
- $\odot_f \subseteq \text{vars}(\varphi)^2 \times \text{vars}(\varphi)^2$ is a symmetric relation called *must-f-path-disjointness*.

Except \odot_f , the components of $G[\varphi]$ represent atomic formulae—equalities, disequalities, pointers, and paths (i.e., list segments)—holding within all models of φ . The fact that $(x_1, y_1) \odot_f (x_2, y_2)$ states that, in all models of φ , the domains of f -paths from x_1 to y_1 and from x_2 to y_2 are disjoint.

To compute the SL-graph $G[\varphi]$, we define some auxiliary notation. We define G_\emptyset to be an SL-graph where all the relations are empty. We write $G \triangleleft \{x_i \bowtie_i y_i\}_{i \in I}$ to denote the SL-graph G' which is the same as G with the elements $x_i \bowtie_i y_i$ for $i \in I$ added to the corresponding relations. We use \sqcup and \sqcap as a component-wise union and intersection of SL-graphs, respectively. We define the disjoint union of SL-graphs as:

$$\begin{aligned} G_1 \boxplus G_2 &= (G_1 \sqcup G_2) \\ &\triangleleft \{x \oplus y \mid x \in \text{alloc}(G_1), y \in \text{alloc}(G_2), \text{ and } (x \text{ is not nil or } y \text{ is not nil})\} \\ &\triangleleft \{e_1 \odot_f e_2 \mid f \in \text{Field}, e_1 \in \text{paths}_f(G_1), \text{ and } e_2 \in \text{paths}_f(G_2)\}. \end{aligned}$$

Here, $\text{paths}_f(G)$ is defined as $\ominus_f \cup \ominus_{\bar{f}}$, and the set of must-allocated variables is $\text{alloc}(G) = \{x \mid \exists y, f. x \ominus_f y \text{ or } (x \ominus_{\bar{f}} y \text{ and } x \oplus y)\} \cup \{\text{nil}\}$ (nil is added for technical reasons). We further assume that all operations on SL-graphs (\triangleleft , \sqcup , \sqcap , and \boxplus) preserve relational properties (symmetry, transitivity, etc.) of the components of SL-graphs by computing the corresponding closures after the operation is performed. We compute the SL-graph $G[\varphi]$ as follows.

$$\begin{aligned} G[x = y] &= G_\emptyset \triangleleft \{x \ominus y\} & G[x \mapsto \langle f_i : f_i \rangle_{i \in I}] &= G_\emptyset \triangleleft \{x \ominus_{f_i} f_i\}_{i \in I} \\ G[x \neq y] &= G_\emptyset \triangleleft \{x \oplus y\} & G[\text{sls}(x, y)] &= G_\emptyset \triangleleft \{x \ominus_n y\} \\ G[\psi_1 \wedge \neg \psi_2] &= G[\psi_1] & G[\text{dls}(x, y, x', y')] &= G_\emptyset \triangleleft \{x \ominus_n y, x' \ominus_p y'\} \\ G[\psi_1 \wedge \psi_2] &= G[\psi_1] \sqcup G[\psi_2] & G[\text{nls}(x, y, z)] &= G_\emptyset \triangleleft \{x \ominus_n z, x \ominus_t y\} \\ G[\psi_1 \vee \psi_2] &= G[\psi_1] \sqcap G[\psi_2] & G[\psi_1 * \psi_2] &= G[\psi_1] \boxplus G[\psi_2] \end{aligned}$$

Observe that we only approximate dls and nls. After the construction is finished, we apply the following rules for matching of pointers and for detection of inconsistencies.

$$\frac{x_1 \ominus_f y_1 \quad x_2 \ominus_{\bar{f}} y_2 \quad x_1 \ominus x_2}{y_1 \ominus y_2} \quad (\mapsto\text{-match}) \qquad \frac{x \ominus y \quad x \oplus y}{\varphi \text{ is unsat}} \quad (\text{contradiction})$$

Tighter location bounds. Using SL-graphs, we can slightly improve the location bound from Section 4 by considering equivalence classes of \ominus instead of individual variables (this can be also used to refine the later described path bound computation) and by defining $\|x\| = 1$ if x is a must-pointers, i.e., $x \ominus_f y$ for some f and y .

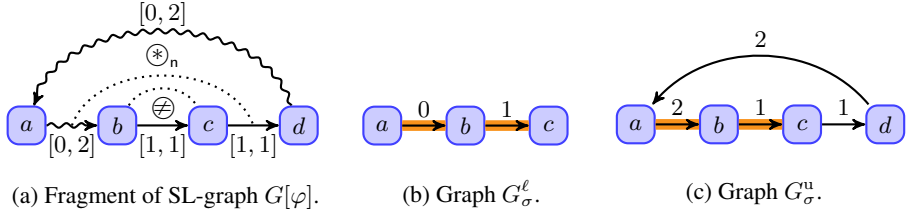


Fig. 3: An illustration of the bound computation for the path σ from a to c on a fragment of SL-graph of $\varphi \triangleq (\text{sls}(a, b) * b \mapsto c * c \mapsto d * \text{sls}(d, a)) \wedge \neg(\text{sls}(a, c) * \text{sls}(c, a))$. The highlighted edges denote the paths used to determine the bound $[1, 3]$.

Path bounds. We now fix an f-path σ from x^S to y and show how to compute an interval $[\ell, u]$ that gives bounds on its length. The computation of the path bounds runs in two steps. In the first step, we compute an initial bound $[\ell_e^0, u_e^0]$ for each edge $e \in \text{paths}_f(G)$. If e is a pointer edge, its bound is given as $[1, 1]$. For a path edge $e = (a, b)$, we define $\ell_e^0 = 1$ if $a \oplus b$ and 0 otherwise; while u_e^0 is defined as $\text{bound}_S(\varphi) - \sum_{v \in V} \|v\|$ where $V = \{v \in \text{vars}_S(\varphi) \mid v \text{ is not } x \text{ and } \exists u. (v, u) \otimes_f (x, y)\}$. This way, we exclude from the computation of the initial upper bound the source v of each path disjoint with σ and all locations possibly allocated in a chunk with the root v . Note that it can be the case that the actual size of this chunk has a lesser size than $\|v\|$, but this means that we were too conservative when computing the global location bound and can decrease the path bound by the same number anyway.

In the second phase, we compute the bounds of the path σ using initial bounds from the first step. The computation is based on two weighted directed graphs derived from the SL-graph G : G_σ^u for the upper bound and G_σ^ℓ for the lower bound (in both cases, the vertices are implicitly given as $\text{vars}(\varphi)$, and the edge weight of an edge e is given by u_e^0 and ℓ_e^0 computed in the previous step, respectively):

$$\begin{aligned} G_\sigma^u &= \{a \rightarrow b \mid (a, b) \in \text{paths}_f(G)\}, \\ G_\sigma^\ell &= \{a \rightarrow b \mid (a \ominus_f b \text{ and } a \oplus_f y) \text{ or} \\ &\quad (a \ominus_f b \text{ and } \exists w. \text{nonempty}(y, w) \text{ and } (y, w) \otimes_f (a, b))\}. \end{aligned}$$

Here, the condition $\text{nonempty}(y, w)$ states that a directed SL-graph edge (y, w) is non-empty which holds if either $y \ominus_f w$, or when $y \sim_f w$ and $y \oplus_f w$.

Intuitively, the upper bound u is computed as the length of the shortest path from x to y in G_σ^u . Since f-paths are uniquely determined, we know that no path can be longer than the shortest one, and thus u is indeed a correct upper bound. The lower bound ℓ is computed as the length of the longest path starting from x (ending anywhere) in G_σ^ℓ . By construction, G_σ^ℓ contains only those edges for which one can prove that they cannot contain y in their domains. A path from x of a length ℓ therefore implies that x cannot reach y in less than ℓ steps, and thus ℓ is indeed a correct lower bound.

Example. We demonstrate the path bound computation in Fig. 3, which shows a fragment of the SL-graph of a formula φ (it shows only those \otimes_n edges that are relevant in our example) and the graphs G_σ^ℓ and G_σ^u for the path σ from a to c . We have that $\|b\| = \|c\| = 1$ and $\|a\| = \|d\| = 2$. This gives us the location bound, which is 6. In the first phase, we compute the initial bound $[0, 2]$ for paths of the predicates $\text{sls}(a, b)$ and $\text{sls}(d, a)$ because both of them are disjoint with all the other paths in $G[\varphi]$. In the second phase, we get the bound for σ equal to $[1, 3]$ instead of the default bound $[0, 6]$.

7 Experimental Evaluation

We have implemented the proposed decision procedure in a new solver called ASTRAL⁷. ASTRAL is written in OCaml and can use multiple backend SMT solvers. With the encoding presented in Section 5, it can use either CVC5 supporting set theory directly [1] or Z3 supporting it by a reduction to the extended theory of arrays [22]. We have also developed an alternative encoding in which both locations and location sets are represented as bitvectors. The bitvector encoding differs only in expressing set operations on the level of bitvectors with additional axioms ensuring that all locations “can fit” into sets encoded by the bitvectors (for details, see [9]). With the bitvector encoding, a backend solver only needs to support theories of bitvectors and arrays, which are both standard and supported by many other SMT solvers. Another advantage is that the quantification on bitvectors seems to perform significantly better than on sets.

In our experiments, if we do not say explicitly which encoding and solver is used, we use the bitvector encoding and BITWUZLA [25] as the backend solver, which we found to be the best performing combination. We set a limit for the method SatEnum to 64 footprints. If this limited is exceeded, we dynamically switch to SatQuantif. We use path quantifiers when the path bound is at most half of the domain bound. These are design choices that can be revisited in the future.

All experiments were run on a machine with 2.5 GHz Intel Core i5-7300HQ CPU and 16 GiB RAM, running Ubuntu 18.04. The timeout was set to 60 s and the memory limit to 1 GB. Our experiments were conducted using BENCHEXEC [4], a framework for reliable benchmarking.

7.1 Entailments of Symbolic Heaps

In the first part of our evaluation, we focus on formulae from the symbolic heap fragment which is frequently used by verification tools and for which there exist many dedicated solvers. We therefore do not expect to outperform the best existing tools but rather to obtain a comparison with other translation-based decision procedures.

In Table 1a, we provide results for the category QF.SHLID.ENTL (entailments with SLLs). We divide the category into two subsets: verification conditions (which are simpler) and more complex artificially generated formulae “*bolognesa*” and “*clones*” from [23]. During the experiments, we found out that several “cloned” entailments contain root variables on the right-hand side of the entailment that do not appear on the left-hand side, making the entailment trivially invalid when its left-hand side is satisfiable. For a few hard clone instances, this makes a problem for ASTRAL as it cannot use the path bound computation as such roots do not appear in the SL-graph. We have therefore implemented a heuristic that detects entailments $\varphi \models \psi$ that can be reduced to satisfiability of φ . Since this is a benchmark-specific heuristic, we present also the version without this heuristic (ASTRAL*) in Table 1a. The optimised version of ASTRAL is able to solve all the formulae being faster than other translation-based solvers GRASSHOPPER⁸ and SLOTH. For illustration, the table further contains the second best solver in the latest edition of SL-COMP, S2S⁹.

⁷ <https://github.com/TDacic/Astral>

⁸ Since GRASSHOPPER is not an solver but a verification tool, we encode the entailment checking as a verification of an empty program.

⁹ We had technical issues running the winner ASTERIX [24]. The difference between those tools is, however, negligible.

Table 1: Experimental results for formulae from SL-COMP. The columns are: solved instances (OK), out of time/memory (RO), instances on which ASTRAL wins—ASTRAL can solve it and the other solver not or ASTRAL solves it faster (WIN), instances solved in the time limits of 0.1 s and 1 s, and the total time for solved instances in seconds.

(a) Results for the category QF_SHLS_ENTL.

Solver	Verification conditions (86)						bolognesa+clones (210)					
	OK	RO	WIN	<0.1	≤1	Total time	OK	RO	WIN	<0.1	≤1	Total time
ASTRAL	86	0	-	84	86	4.62	210	0	-	68	169	202.91
ASTRAL*	86	0	42	83	86	4.64	195	15	88	64	150	408.48
GRASSHOPPER	86	0	70	52	86	8.65	203	7	148	60	87	1229.35
S2S	86	0	5	86	86	2.08	210	0	3	203	210	8.18
SLOTH	64	3	86	0	28	235.28	70	140	210	0	50	149.42

(b) Results for a subset of the category QF_SHLID_ENTL.

Solver	Doubly-linked lists (17)						Nested singly-linked lists (19)					
	OK	RO	WIN	<0.1	≤1	Total time	OK	RO	WIN	<0.1	≤1	Total time
ASTRAL	17	0	-	11	17	2.72	19	0	-	3	9	86.93
GRASSHOPPER	17	0	16	3	15	7.53	-	-	-	-	-	-
HARRSH	17	0	17	0	0	95.18	14	5	18	0	0	183.01
S2S	17	0	0	17	17	0.15	19	0	0	19	19	0.43
SONGBIRD	11	5	14	5	9	13.39	11	5	8	4	11	1.38

In Table 1b, we provide results for a subset of the category QF_SHLID_ENTL (entailments with linear inductive definitions from which we selected DLLs and NLLs) for ASTRAL and three best-performing solvers competing in the latest edition of SL-COMP—S2S, SONGBIRD (in the version with automated lemma synthesis called SLS), and HARRSH. We also include GRASSHOPPER which supports DLLs only. Except S2S which solves almost all formulae virtually immediately, ASTRAL is the only one able to solve all the formulae in the given time limit.

7.2 Experiments on Formulae Outside of the Symbolic Heap Fragment

For formulae outside of the symbolic heap fragment and its top-level boolean closure, there are currently no existing benchmarks. For now, we therefore limit ourselves to randomly generated but extensive sets of formulae. In the future, we would like to develop a program analyser using symbolic execution over BSL and make more careful experiments on realistic formulae.

We first focus on the fragment with guarded negations but without inductive predicates, on which we can compare ASTRAL with CVC5. We have prepared a set of 1000 entailments of the form $\varphi \models \psi$ which are generated as random binary trees with depth 8 over 8 variables with the only atoms being pointer assertions. To reduce the number of trivial instances, we only generated formulae for which $\text{vars}(\psi) \subseteq \text{vars}(\varphi)$ and ASTRAL cannot deduce contradiction from their SL-graphs. To avoid any suspicion that the difference is caused by better performance of the backend solver rather than the design of our translation, we used ASTRAL with the CVC5 backend and direct set

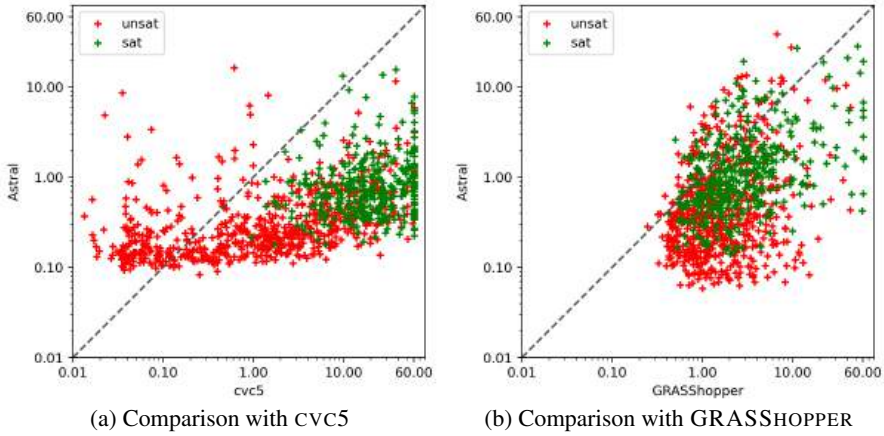


Fig. 4: A comparison of ASTRAL with CVC5 and GRASSHOPPER on randomly generated formulae. Times are in seconds, axes are logarithmic. The timeout was set to 60 s.

encoding (with BITWUZLA and bitvector encoding, our results would be even better). The results are given in Fig. 4a and suggest that our treatment of guarded negations really brings a better performance—ASTRAL can solve all the instances and almost all of them under 10 seconds. On the other hand, CVC5 timed out in 61 cases and is usually slower than ASTRAL, in particular on satisfiable formulae which represent invalid entailments.

In the second experiment, we compared our solver with GRASSHOPPER on the fragment which it supports, i.e., arbitrary nesting of conjunctions and disjunctions. We again generated 1000 entailments, this time with depth 6, 6 variables and with atoms being singly-linked lists (with 20 % probability) or pointer-assertions. The results are given in Fig. 4b. ASTRAL ran out of memory in 5 cases, and GRASSHOPPER timed out in 10 cases. In summary, ASTRAL is faster on more than 80 % of the formulae with an almost 3 times lesser running time.

Finally, to illustrate that ASTRAL can indeed handle formulae out of the fragments of all the other mentioned tools, we apply it on an entailment query that involves the formula mentioned at the end of the introduction: $((\text{sls}(x, y) \wedge \neg(\text{sls}(x, z) * \text{sls}(z, y))) * y \mapsto z) \models \text{sls}(x, z)$, converted to an unsatisfiability query. ASTRAL resolves the query in 0.12 s. Note that without the requirement $\neg(\text{sls}(x, z) * \text{sls}(z, y))$, the entailment does not hold as a cycle may be closed in the heap.

8 Conclusions and Future Work

We have presented a novel decision procedure based on a small-model property and translation to SMT. Our experiments have shown very promising results, especially for formulae with rich boolean structure for which our decision procedure outperforms other approaches (apart from being able to solve more formulae).

In the future, we would like to extend our approach with some class of user-defined inductive predicates, with more complex spatial connectives such as septractions and/or magic wands, consider a lazy and/or interactive translation instead of the current eager approach, and try ASTRAL within some SL-based program analyser.

References

1. Bansal, K., Barrett, C., Reynolds, A., Tinelli, C.: A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In: IJCAR (2017)
2. Batz, K., Fesefeldt, I., Jansen, M., Katoen, J.P., Keßler, F., Matheja, C., Noll, T.: Foundations for Entailment Checking in Quantitative Separation Logic. In: ESOP (2022)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In: FSTTCS 2004. LNCS, vol. 3328 (2004)
4. Beyer, D., Löwe, S., Wendler, P.: Reliable Benchmarking: Requirements and Solutions. *International Journal on Software Tools for Technology Transfer* **21** (2017)
5. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A Generic Cyclic Theorem Prover. In: APLAS. LNCS, vol. 7705 (2012)
6. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM* **58**(6) (2011)
7. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In: FST TCS (2001)
8. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable Reasoning in a Fragment of Separation Logic. In: CONCUR. LNCS, vol. 3901 (2011)
9. Dacík, T., Rogalewicz, A., Vojnar, T., Zuleger, F.: Deciding Boolean Separation Logic via Small Models. Tech. rep. (10 2023), <https://zenodo.org/records/10012893>
10. Echenim, M., Iosif, R., Peltier, N.: The Bernays-Schönfinkel-Ramsey Class of Separation Logic with Uninterpreted Predicates. *ACM Transactions on Computational Logic* **21** (2019)
11. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: Compositional Entailment Checking for a Fragment of Separation Logic. In: APLAS (2014)
12. Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T., Zuleger, F.: Low-level bi-abduction. In: ECOOP 2022. LIPIcs, vol. 222, pp. 19:1–19:30 (2022)
13. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding Entailments in Inductive Separation Logic with Tree Automata. In: ATVA (2014)
14. Iosif, R., Zuleger, F.: Expressiveness results for an inductive logic of separated relations. In: Pérez, G.A., Raskin, J. (eds.) CONCUR. LIPIcs, vol. 279, pp. 20:1–20:20 (2023). <https://doi.org/10.4230/LIPICS.CONCUR.2023.20>, <https://doi.org/10.4230/LIPICS.CONCUR.2023.20>
15. Ishtiaq, S., O'Hearn, P.: Separation and Information Hiding. In: Proc. of POPL'01. ACM (2001)
16. Katelaan, J., Jovanovic, D., Weissenbacher, G.: A Separation Logic with Data: Small Models and Automation. In: IJCAR (2018)
17. Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Harrsh: A Tool for Unied Reasoning about Symbolic-Heap Separation Logic. In: LPAR-22 Workshop and Short Paper Proceedings. vol. 9 (2018)
18. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape Analysis via Second-Order Bi-Abduction. In: Proc. of CAV'14. LNCS, vol. 8559. Springer (2014)
19. Le, Q.L.: Compositional Satisfiability Solving in Separation Logic. In: VMCAI. LNCS, vol. 12597 (2021)
20. Le, Q.L., Le, X.B.D.: An Efficient Cyclic Entailment Procedure in a Fragment of Separation Logic. In: FoSSaCS (2023)
21. Matheja, C., Pagel, J., Zuleger, F.: A Decision Procedure for Guarded Separation Logic Complete Entailment Checking for Separation Logic with Inductive Definitions. *ACM Trans. Comput. Logic* **24**(1) (2023)
22. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD (2009)

23. Navarro Pérez, J.A., Rybalchenko, A.: Separation Logic + Superposition Calculus = Heap Theorem Prover. In: PLDI (2011)
24. Navarro Pérez, J.A., Rybalchenko, A.: Separation Logic Modulo Theories. In: APLAS. LNCS, vol. 8301 (2013)
25. Niemetz, A., Preiner, M.: Bitwuzla. In: CAV. LNCS, vol. 13965 (2023)
26. Pagel, J., Zuleger, F.: Strong-separation logic. *ACM Trans. Program. Lang. Syst.* **44**(3), 16:1–16:40 (2022). <https://doi.org/10.1145/3498847>, <https://doi.org/10.1145/3498847>
27. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic Using SMT. In: CAV (2013)
28. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic with Trees and Data. In: CAV (2014)
29. Reynolds, A., Iosif, R., King, T.: A Decision Procedure for Separation Logic in SMT. In: ATVA (2016)
30. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (2002)
31. Santos, J., Maksimovic, P., Ayoun, S.E., Gardner, P.: Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In: Proc. of PLDI'20. ACM (2020)
32. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs (extended version). *Int. J. Softw. Tools Technol. Transf.* **22**(6), 709–728 (2020). <https://doi.org/10.1007/s10009-020-00559-y>
33. Ta, Q.T., Le, T.C., Khoo, S.C., Chin, W.N.: Automated Lemma Synthesis in Symbolic-Heap Separation Logic. In: POPL (2018)
34. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable Shape Analysis for Systems Code. In: Proc. of CAV'08. LNCS, vol. 5123. Springer (2008)

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.





Asynchronous Subtyping by Trace Relaxation

Laura Bocchi¹ , Andy King¹ , and Maurizio Murgia²

¹ University of Kent, Canterbury, CT2 7NZ, UK

{l.bocchi,a.m.king}@kent.ac.uk

² Gran Sasso Science Institute, 67100 L'Aquila, AQ, Italy

maurizio.murgia@gssi.it

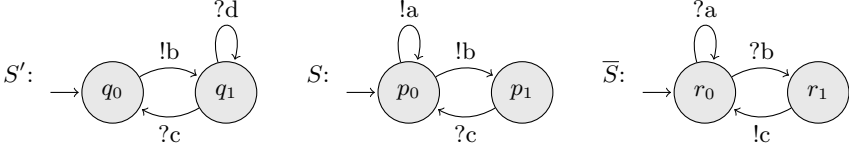
Abstract. Session subtyping answers the question of whether a program in a communicating system can be safely substituted for another, when their communication behaviours are described by session types. Asynchronous session subtyping is undecidable, hence the interest in devising sound, although incomplete, subtyping algorithms. State-of-the-art algorithms are formulated in terms of a data-structure called input trees. We show how input trees can be replaced by sets of traces, which opens up opportunities for applying techniques abstract interpretation techniques to the problem of asynchronous session subtyping. Sets of traces can be relaxed (enlarged) whilst still allowing subtyping to be observed, and one can choose relaxations that can be finitely represented, even when the input trees are arbitrarily large. We instantiate this strategy using regular expressions and show that it allows subtyping to be mechanically proven for communication patterns that were previously out of reach.

Keywords: asynchrony, session subtyping, automata, abstract interpretation

1 Introduction

Protocols, which are used to communicate and orchestrate activity in distributed systems, are notoriously difficult to write and understand. Session types [23, 34] have thus been proposed for specifying protocol interaction and automatically checking whether an implementation conforms to its specification. Session types extend data types to describe communication behaviour, and express the behaviour of units of design (sessions) in terms of which types of messages can be sent or received, and in what order. They have been integrated into mainstream languages and proved to be a powerful tool for static [25, 26, 28, 31, 32] and dynamic [1, 2] verification as well as API generation [24, 30].

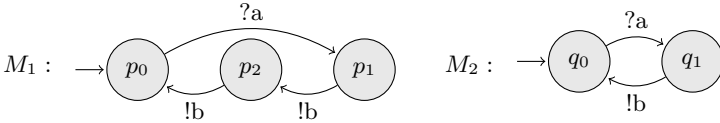
Session Subtyping A fundamental problem in the application of session types is checking whether the implementation of one component in a distributed system can be substituted for another, without violating an overarching protocol. This problem can be formulated as *session subtyping* [11, 18, 20, 21], which is a preorder relation on session types: S' is a sub-type of S , written $S' \leq S$, if a program with type S can be safely substituted by a program with type S' . Consider S and S' below:



S and S' are expressed in automata notation where $!a$ (resp. $?a$) denotes a send (resp. receive) action on channel a . S models a process, which in state p_0 , can repeatedly request a service a , or request b and then receive a confirmation c .

The overarching protocol is defined, in a binary (client-server) session, as the parallel composition of S with its dual \bar{S} , written $S \mid \bar{S}$. The dual \bar{S} is obtained by swapping each send action with a corresponding receive action and vice versa. Due to syntactic constraints posed by session types [23], $S \mid \bar{S}$ enjoys a number of key properties (e.g., deadlock freedom, communication safety). A process behaving as S can be safely substituted with another behaving as S' that has less sends (e.g. the absent $!a$) and more receives (e.g. the additional $?d$). This notion of substitutability is co-variant on send actions and contra-variant on receive actions, and preserves the key properties in protocol $S' \mid \bar{S}$.

We focus on *asynchronous session subtyping* (async subtyping for short) as asynchronous communications (over FIFO channels) are key in distributed systems and languages such as Go and Rust. Async subtyping, however, is undecidable [6, 27]. We focus on *asynchronous session subtyping* (async subtyping for short) as asynchronous communications (over FIFO channels) are key in distributed systems and languages such as Go and Rust. Async subtyping, however, is undecidable [6, 27] so the search is on for sound algorithms which are sufficiently robust to prove subtyping in the majority of cases. Given a candidate subtype and a supertype, the subtyping problem can be viewed as a simulation game in which the supertype is required to mirror any input and output action performed by the subtype. Since communication is asynchronous, the subtype can send early in the sense that the supertype can only realise the same output after some inputs. Consider M_2 below, which models a server producing a news feed ($!b$) on request from a client ($?a$), where M_1 is a candidate subtype for M_2 :



After receiving on a , M_2 can immediately mimic the first send on b of M_1 , but it can only perform the second send on b after receiving another request. The input a is said to guard the output b . One needs to reason about these dependencies to verify that M_2 can follow the actions of M_1 , albeit with (a possibly unbounded number of) send actions being delayed. This is the challenge of asynchronous subtyping. Apart from substitutability, asynchronous subtyping enables protocol optimisation in which receives are postponed, so as to minimise busy waiting for messages [29]. In M_2 , if feed production was more efficient than request processing then it would be better if the server bundled feeds, as in M_1 .

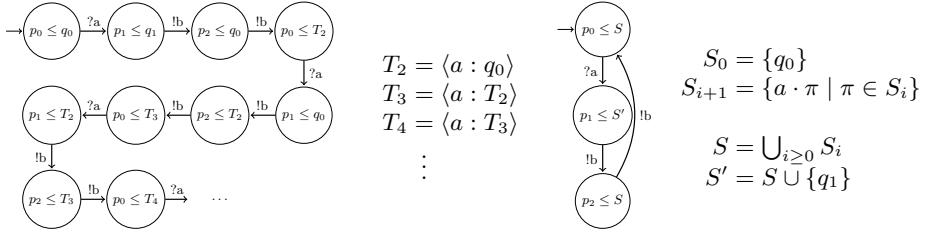


Fig. 1. A simulation tree (left) and collecting simulation graph (right) for M_1 and M_2

Existing techniques The state-of-the-art approach to async subtyping [4, 5] represents a simulation game between the (candidate) subtype and supertype, in its entirety, with a simulation tree. The state of the supertype is modelled using an input tree [4, 5, 11, 10], which records and accumulates input actions which guard outputs. Figure 1 gives the simulation tree for M_1 and M_2 . Simulation commences at $p_0 \leq q_0$ where M_1 and M_2 are in their initial states p_0 and q_0 . The edges in the tree follow the actions of M_1 , with M_2 following along using its input tree. Step $p_0 \leq T_2$ models the scenario where M_1 is in state p_0 but, in M_2 , a second send on b is guarded by a receive on a . Input tree $T_2 = \langle a : q_0 \rangle$ expresses this dependency by recording that M_2 can continue at q_0 , after performing the pending receive on a . As the simulation of M_1 unfolds, however, the input trees for M_2 grow without bound, yielding an infinite simulation tree.

Previous work [4, 5] proposed a multi-step algorithm that computes a simulation tree until violation of a syntactic condition [5, Theorem 3.8] that is formulated in terms of the depth of input trees. The simulation tree is then divided into sub-trees, which are checked against a safety property [5, Definition 3.16]. The sub-trees are then used to generate systems of equations which are solved and checked against a compatibility condition [5, Definition 3.12]. The construction is ingenious, but the length of the proofs [5, p. 14, p. 19-20, p. 22-26] begs the question of whether subtyping can be solved more simply. Furthermore, can a strategy be found that is amenable to independent algorithmic checking? This would explain why subtyping holds, further instilling confidence.

Contribution Our development starts with the observation that an input tree can be represented, without loss of information, as a set of traces: one trace for each branch through the input tree. The rationale behind this encoding is that sets of traces can: (1) be relaxed (enlarged) and (2) be described as regular expressions. As to (1), a trace-based representation allows the subtyping algorithm to relax a set of traces to a strictly larger (possibly infinite) set, whilst still allowing subtyping to be observed. By covering all the sets of traces that arise in a simulation tree with a finite number of trace sets we can fold a simulation tree onto a graph to obtain a tractable (finite) representation. Regarding (2), (possibly infinite) sets of traces can themselves be finitely represented as regular expressions. For example, Figure 1 (right) shows a collecting simulation graph

where the states of M_2 are relaxed to the set of traces S and S' , which can be represented, say, as a^*q_0 and $a^*q_0 + q_1$ respectively. The result is a subtyping algorithm equipped with relaxation and termination machinery which can prove subtyping on more (and more complex) problems than existing methods.

The use of sets of traces separates the proof for correctness of the core algorithm, from the problem of how to finitely represent sets of traces. This separation simplifies the theoretical development. If higher fidelity was required, regular expressions could be replaced with context-free grammars [13]; alternatively the relaxations employed with regular expressions (string widening [12]) can be tuned without revisiting the correctness of the core algorithm.

Synopsis Section 2 introduces (session types as) communicating machines; Section 3 defines asynch subtyping with the formulation in [5] to facilitate comparison and Section 4 gives a sound formulation based on collecting simulation graphs. Section 5 gives an algorithm based on regular expressions and widening over collecting simulation graph, and introduces and evaluates our tool. Conclusion and related work are in Section 6.

2 Preliminaries on Communicating Machines

Let A denote a finite alphabet, and $\mathbb{A} = \{!, ?\} \times A$ denote a finite set of send and receive actions. A communicating machine $M = (Q, q_0, \delta)$ (machine for short) is defined by a finite set of states Q , an initial state $q_0 \in Q$, and a transition relation $\delta \subseteq Q \times \mathbb{A} \times Q$. For a fixed machine $M = (Q, q_0, \delta)$, we write: $q \xrightarrow{w} q'$ iff $(q, w, q') \in \delta$; $q \xrightarrow{w}$ iff there exists q' such that $q \xrightarrow{w} q'$; $q_0 \xrightarrow{w_1, \dots, w_n} q_n$ iff there exist $q_1, \dots, q_{n-1} \in Q$ such that $q_i \xrightarrow{w_{i+1}} q_{i+1}$ for $0 \leq i \leq n-1$.

Given a sequence of labels $\vec{a} = a_1, \dots, a_k$ and a direction $\star \in \{!, ?\}$, we write $\star\vec{a}$ for the sequence of actions $\star a_1, \dots, \star a_k$. The maps $\text{in}_M : Q \rightarrow \wp(A)$ and $\text{out}_M : Q \rightarrow \wp(A)$ are defined: $\text{in}_M(q) = \{a \in A \mid q \xrightarrow{?a}\}$ and $\text{out}_M(q) = \{a \in A \mid q \xrightarrow{!a}\}$. The predicate $\text{send}_M(q)$ holds iff $\text{out}_M(q) \neq \emptyset$ and $\text{recv}_M(q)$ holds iff $\text{in}_M(q) \neq \emptyset$. The predicate $\text{final}_M(q)$ holds iff $\neg \text{send}_M(q)$ and $\neg \text{recv}_M(q)$.

Definition 1 (Session types correspondence). *For a given $M = (Q, q_0, \delta)$, M is deterministic iff $(q, w, q_1), (q, w, q_2) \in \delta$ implies $q_1 = q_2$; M has no mixed states iff $\neg \text{send}_M(q)$ or $\neg \text{recv}_M(q)$ for all $q \in Q$. A session type corresponds [19] to a deterministic machine without mixed states.*

Henceforth we focus on systems of *two* deterministic machines without mixed states, which correspond to *binary* session types. Binary session types describe two-party protocols (e.g., client-server as POP2, SMTP). State-of-the-art asynchronous subtyping algorithms [5] are formulated on binary sessions (each session involving two rather than many participants). We focus on demonstrating how abstraction can be applied to these algorithms and thus, likewise, adopt the binary setting.

Because M is deterministic, the relation δ can be interpreted as a partial function $Q \times \mathbb{A} \rightarrow Q$ defined by $\delta(q, \ell) = q'$ iff $q \xrightarrow{\ell} q'$. Following [5] we introduce the predicate $\text{cycle}_M(!, q)$ to aid the characterisation of orphan messages:

Definition 2. *The predicate $\text{cycle}_M(!, q)$ holds iff there exist $\vec{a} \in A^*$, $\vec{b} \in A^+$ and $q' \in Q$ such that $q \xrightarrow{! \vec{a}} q'$ and $q' \xrightarrow{\vec{b}} q'$.*

The predicate $\text{cycle}_M(!, q)$ thus holds iff from q one can reach, using a possibly empty sequence of send actions, a cycle (from q' to q' itself) of send actions. The predicate $\text{cycle}_M(?, q)$ is defined analogously.

3 Asynchronous Subtyping with Input Trees

We define input trees and asynchronous subtyping, adopting the formulation of [5]. Input trees are defined over the states Q of a supertype. Asynchronous subtyping is then defined in terms of input trees, the trees capturing input accumulation for guarded outputs.

Definition 3. *The set of input trees T_Q over Q is the least set such that: (1) if $q \in Q$ then $q \in T_Q$; (2) if I is an index set, $\forall i \in I. a_i \in A$, $t_i \in T_Q$ and $\forall i, j \in I. i \neq j \implies a_i \neq a_j$ then $\langle a_i : t_i \mid i \in I \rangle \in T_Q$*

An input tree over Q is either a state in Q or an accumulated input. A term of the form $\langle a_i : t_i \mid i \in I \rangle$ represents an accumulated input that presents an options a_i for each $i \in I$, followed by a tree t_i . Note that any input tree of T_Q is necessarily finite. The following definition shows how to build the input tree $\text{inTree}_M(q)$ for a state q of a given machine M , and defines the associated set of leaves $\text{leaf}(t)$ of the input tree t .

Definition 4 (Input tree). *Define $\text{inTree}_M : Q \rightarrow T_Q$ and $\text{leaf} : T_Q \rightarrow \wp(Q)$*

$$\begin{aligned} \text{inTree}_M(q) &= \begin{cases} \perp & \text{if } \text{cycle}_M(?, q) \\ q & \text{else if } \text{in}_M(q) = \emptyset \\ \langle a_i : \text{inTree}_M(\delta(q, ?a_i)) \mid i \in I \rangle & \text{else if } \text{in}_M(q) = \{a_i \mid i \in I\} \end{cases} \\ \text{leaf}(t) &= \begin{cases} \{t\} & \text{if } t \in Q \\ \bigcup \{\text{leaf}(t_i) \mid i \in I\} & \text{else if } t = \langle a_i : t_i \mid i \in I \rangle \end{cases} \end{aligned}$$

The $\text{cycle}_M(?, q)$ condition (also used in [5]) ensures that $\text{inTree}_M(q)$, if defined, is finite. Note that $a_i : \text{inTree}_M(q_i)$ is well-defined in the above. To see why, suppose $\delta(q, a_i) = q_i$. Observe that if $\neg \text{cycle}_M(?, q)$ then $\neg \text{cycle}_M(?, q_i)$. Repeating this argument it follows $\text{inTree}_M(q_i) \neq \perp$, as required.

Example 1 (Running example: input trees and leaves). The machines **14may2** and **14may1** specified in Figure 2 originate from the GitHub repository which accompanies [5]. Henceforth let $N_1 = \mathbf{14may2}$ and $N_2 = \mathbf{14may1}$.

$$\begin{aligned} \text{inTree}_{N_2}(q_0) &= \langle a : \langle a : q_2, c : q_3 \rangle, c : q_5 \rangle & \text{leaf}(\text{inTree}_{N_2}(q_0)) &= \{q_2, q_3, q_5\} \\ \text{inTree}_{N_2}(q_1) &= \langle a : q_2, c : q_3 \rangle & \text{leaf}(\text{inTree}_{N_2}(q_1)) &= \{q_2, q_3\} \\ \text{inTree}_{N_2}(q_i) &= q_i \text{ for all } 2 \leq i \leq 6 & \text{leaf}(q_3) &= \{q_3\} \end{aligned}$$

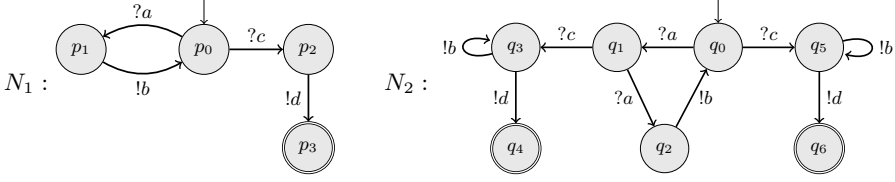


Fig. 2. Communicating machines N_1 (14may2) and N_2 (14may1)

Next, we introduce a substitution θ that we use, in the definition of asynchronous subtyping, to model the accumulation of inputs as simulation unfolds. Input trees are extended at their leaves by the application of a substitution θ .

Definition 5 (Substitution). If $q_i \in Q$ and $t_i \in T_Q$ for all $i \in I$ then $\theta = \{q_i \mapsto t_i \mid i \in I\}$ denotes an operator $T_Q \rightarrow T_Q$ where $\theta(t)$ is the input tree obtained by simultaneously substituting each occurrence of q_i in t with t_i .

In Definition 6 we introduce the notion of an async subtyping relation between states of a candidate subtype and input trees of a supertype. We follow [11] and, like [5], adopt the conventional orphan-free version of asynchronous subtyping [7, Definition 2.4] adapted to the setting of communicating machines:

Definition 6. An async subtyping relation for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is a binary relation $\mathcal{R} \subseteq P \times T_Q$ such that $(p, t) \in \mathcal{R}$ implies:

1. if $\text{final}_{M_1}(p)$ then $t = q$ for some $q \in Q$ and $\text{final}_{M_2}(q)$
2. if $\text{recv}_{M_1}(p)$ then
 - (a) if $t = q$ for some $q \in Q$ then $\text{recv}_{M_2}(q)$ and if $q \xrightarrow{?a} q'$ there exist $p \xrightarrow{?a} p'$ and $(p', q') \in \mathcal{R}$
 - (b) if $t = \langle a_i : t_i \mid i \in I \rangle$ then for all $i \in I$ there exist $p \xrightarrow{?a_i} p'$ and $(p', t_i) \in \mathcal{R}$
3. if $\text{send}_{M_1}(p)$ then:
 - (a) if $t = q$ for some $q \in Q$ and $\text{send}_{M_2}(q)$ then if $p \xrightarrow{!a} p'$ there exist $q \xrightarrow{!a} q'$ and $(p', q') \in \mathcal{R}$
 - (b) otherwise if $\text{leaf}(t) = \{q_i \mid i \in I\}$ then
 - i. $\neg \text{cycle}_{M_1}(!, p)$
 - ii. $t_i = \text{inTree}_{M_2}(q_i) \neq \perp$ for all $i \in I$
 - iii. if $p \xrightarrow{!a} p'$ and $\theta = \{q \mapsto q' \mid q \in Q, q \xrightarrow{!a} q'\}$ then $\text{leaf}(t_i) \subseteq \text{dom}(\theta)$ for all $i \in I$ and $(p', \kappa(t)) \in \mathcal{R}$ where $\kappa = \{q_i \mapsto \theta(t_i) \mid i \in I\}$

Case (1) is self-explanatory. Case (2) is for input actions in M_1 and realises contra-variance with respect to inputs. Case (2.a) applies when the states p and q are in sync, whereas case (2.b) applies when an accumulated input a_i in M_2 is consumed by a corresponding input action of M_1 . In case (2.a), condition $\text{recv}_{M_2}(q)$ ensures that the guarded clause $q \xrightarrow{?a} q'$ does not hold vacuously. Case (3) is for output actions in M_1 and implements output co-variance.

Case (3.a) applies when M_1 and M_2 are in sync, while case (3.b) is for accumulated inputs. The negated cycle_{M_1} of clause (3.b.i) predicate mirrors [5] and prevents orphan messages, ensuring that accumulated inputs are eventually considered. Clause (3.b.ii) was implicit in [5] but is used in the proofs for structuring, and is thus made explicit. Clause (3.b.iii) ensures that if p in M_1 can send, then every leaf of the corresponding input tree t in M_2 can make a matching send action.

Definition 7 (Async Subtyping). $M_1 = (P, p_0, \delta_1)$ is an (async) subtype of $M_2 = (Q, q_0, \delta_2)$, written $M_1 \leq M_2$, iff there exists an async subtyping relation $\mathcal{R} \subseteq P \times T_Q$ for M_1 and M_2 such that $(p_0, q_0) \in \mathcal{R}$.

4 Asynchronous Subtyping with Input Traces

Simulation trees [5] provide a foundation for checking subtyping, but because their branches can grow arbitrarily long, they are not tractable in themselves. To obtain a model which is amenable to abstraction, we substitute an input tree with a set of input traces. Sets of input traces can be easily relaxed by adding more input traces, which is key to deriving a finite alternative representation.

Definition 8 (Input Traces). Given a fixed alphabet A and a set of states Q , input traces (traces for short) are words formed from the alphabet A (which are ranged over by π) followed by a state in Q : $\text{Tr}_Q = \{\pi \cdot q \mid \pi \in A^*, q \in Q\}$. The empty word is denoted ϵ .

The development begins by lifting a simulation tree to sets of traces, a construction which itself requires some set-level auxiliary operations:

Definition 9 (Traces of an input tree). The set of traces of an input tree is given by the map $\text{tr} : T_Q \cup \{\perp\} \rightarrow \wp(\text{Tr}_Q)$ defined by:

$$\text{tr}(t) = \begin{cases} \emptyset & \text{if } t = \perp \\ \{t\} & \text{if } t \in Q \\ \{a_i \cdot \pi \mid \pi \in \text{tr}(t_i), i \in I\} & \text{if } t = \langle a_i : t_i \mid i \in I \rangle \end{cases}$$

Example 2 (Running example: traces). Continuing with N_1 and N_2 of Example 1 (Figure 2), $\text{tr}(\text{inTree}_{N_2}(q_0)) = \{aaq_2, acq_3, cq_5\}$ and $\text{tr}(\text{inTree}_{N_2}(q_1)) = \{aq_2, cq_3\}$.

4.1 Collecting simulation

A (collecting) simulation tree is formulated in terms of a (collecting) simulation relation, defined below. The term collecting has been chosen to resonate with abstract interpretation [15] where a semantics is lifted to operate on sets of data points (to give a so-called collecting semantics) which provides a semantic substrate for synthesising an algorithm.

$$\begin{array}{c}
\frac{\text{in}_{M_2}(q) \subseteq \text{in}_{M_1}(p) \quad q \xrightarrow{?a} q'}{p \leq q \xrightarrow{?a} \delta_{M_1}(p, ?a) \leq q'} [\text{Recv}] \quad \frac{\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q) \quad p \xrightarrow{!a} p'}{p \leq q \xrightarrow{!a} p' \leq \delta_{M_2}(q, !a)} [\text{Send}] \\
\\
\frac{a \in \text{in}_{M_1}(p)}{p \leq a \cdot \pi \xrightarrow{?a} \delta_{M_1}(p, ?a) \leq \pi} [\text{RecvTr}] \quad \frac{\begin{array}{c} \neg \text{cycle}_{M_1}(!, p) \\ \text{tr}(\text{inTree}_{M_2}(q)) = \{\phi_i \cdot q_i \mid i \in I\} \quad k \in I \\ \forall i \in I : \text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q_i) \end{array} \quad p \xrightarrow{!a} p'}{p \leq \phi \cdot q \xrightarrow{!a} p' \leq \phi \cdot \phi_k \cdot \delta_{M_2}(q_k, !a)} [\text{SendTr}] \\
\\
\frac{\forall \pi \in S : \exists b \in A : p \leq \pi \xrightarrow{?b} \quad S_a = \{\pi' \mid \pi \in S, p \leq \pi \xrightarrow{?a} p' \leq \pi'\} \neq \emptyset}{p \leq S \xrightarrow{?a} p' \leq S_a} [\text{RecvSet}] \\
\\
\frac{\forall \pi \in S : p \leq \pi \xrightarrow{!a} \quad S_a = \{\pi' \mid \pi \in S, p \leq \pi \xrightarrow{!a} p' \leq \pi'\} \neq \emptyset}{p \leq S \xrightarrow{!a} p' \leq S_a} [\text{SendSet}]
\end{array}$$

Fig. 3. Rules for trace-based asynchronous subtyping

Definition 10 (Collecting simulation). *The collecting simulation relation of two machines $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is the least 5-place relation $\hookrightarrow \in P \times \wp(\text{Tr}_Q) \times \mathbb{A} \times P \times \wp(\text{Tr}_Q)$, satisfying the rules in Figure 3, where $p \leq S \xrightarrow{\ell} p' \leq S'$ abbreviates $(p, S, \ell, p', S') \in \hookrightarrow$.*

In Figure 3, rules Recv and RecvTr collectively realise the second case of Definition 6: rule Recv realises case (2.a) for interactions in sync, and RecvTr realises case (2.b) that consumes an accumulated input. The contra-variance of receive manifests as $\text{in}_{M_2}(q) \subseteq \text{in}_{M_1}(p)$ in Recv and $a \in \text{in}_{M_1}(p)$ in RecvTr. Rules Send and SendTr realise case (3.a) and case (3.b) of Definition 6, respectively. In these rules, the co-variance of send appears as premise $\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q)$ in Send and $\forall i \in I : \text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q_i)$ in SendTr. In rule SendTr, the $\text{leaf}(t_j) \subseteq \text{dom}(\theta)$ condition in case (3.b.) follows from the premise $\text{out}_{M_1}(p) \subseteq \text{out}_{M_2}(q_i)$ for all $i \in I$. To see this, let $q \in \text{leaf}(t_j)$ for some $j \in J$. Since $p \xrightarrow{!a} p'$, $a \in \text{out}_{M_1}(p)$ thus $a \in \text{out}_{M_2}(q)$ therefore $q \in \text{dom}(\theta)$.

The absence of mixed states (Definition 1) ensures that if both Send and SendTr are applicable then the traces which result coincide. The force of this is that clause ‘otherwise if ...’ of Definition 6(3.b) can be simplified to ‘if ...’ (so there is no need to prioritise the application of Send over SendTr). The current formulation of Definition 6(3.b) was chosen to align with that used in [5].

Rules RecvSet and SendSet lift subtyping from traces to sets of traces. In RecvSet, the first premise specifies a covering requirement: that a receive is possible for each trace of S . The second premise prescribes a grouping requirement: for a given receive action $?a$, the second precondition accumulates all those traces which can be derived by receiving on a . The requirement $S_a \neq \emptyset$ ensures that a non-empty subset of S contributes to S_a . The $S_a \neq \emptyset$ requirement, which

likewise shows up in **SendSet**, also inhibits meaningless transitions of the form $p \leq \emptyset \xrightarrow{?a} p' \leq \emptyset$ and $p \leq \emptyset \xrightarrow{!a} p' \leq \emptyset$, which would otherwise hold vacuously.

For any given $p \leq S$, relaxing S to T , can result in either $p \leq T$ becoming stuck, or a move that preserves the inclusion of traces. To formulate this property, let $p \leq T \not\xrightarrow{\ell}$ denote the absence of a transition of the form $p \leq T \xrightarrow{\ell} p' \leq T'$.

Proposition 1 (Monotonicity). *Let $T \subseteq S \subseteq Tr_Q$ and $\ell \in \mathbb{A}$. Then if $p \leq T \xrightarrow{\ell} p' \leq T'$ either: $p \leq S \not\xrightarrow{\ell}$ or $p \leq S \xrightarrow{\ell} p' \leq S'$ where $T' \subseteq S'$.*

4.2 Collecting simulation trees and graphs

First, we provide an infinite model for collecting simulation using collecting simulation trees, that is an alternative presentation of simulation trees [5] where we represent the state of a supertype as a set of traces rather than an input tree.

Definition 11 (Collecting simulation (sim) tree). *A collecting sim tree for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is a labelled tree $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ where $\xrightarrow{\ell}_t \subseteq N \times N$ is a tree rooted at n_0 and $\mathcal{L} : N \rightarrow P \times \wp(Tr_Q)$ such that:*

1. $\mathcal{L}(n_0) = (p_0, \{q_0\})$
2. if $p \leq S \xrightarrow{\ell} p' \leq S'$ and $\mathcal{L}(n) = (p, S)$ then $n \xrightarrow{\ell}_t n'$ for some $n' \in N$ such that $\mathcal{L}(n') = (p', S')$
3. if $n \xrightarrow{\ell}_t n'$ and $\mathcal{L}(n) = (p, S)$ then $\mathcal{L}(n') = (p', S')$ such that $p \leq S \xrightarrow{\ell} p' \leq S'$

Case (2) above ensures that a collecting sim tree enumerates *all* the transitions of $\xrightarrow{\ell}$ whereas case (3) ensures that the tree *only* enumerates $\xrightarrow{\ell}$ transitions. Note that a collecting sim tree is unique up to tree isomorphism.

Theorem 1 shows that subtyping can be expressed in terms of successful branches (Definition 12) of collecting sim trees.

Definition 12 (branches). *A branch of a collecting sim tree $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ is a (possibly infinite) sequence $n_0, n_1, \dots \subseteq N$ such that $n_i \xrightarrow{\ell}_t n_{i+1}$ for all consecutive n_i, n_{i+1} . A complete branch of the collecting sim tree is a branch which is not a strict prefix of another branch of the collecting sim tree. A successful branch is a complete branch which is either infinite or whose last node n is labelled $\mathcal{L}(n) = (p, F)$ with $F \subseteq Q$, $\text{final}_{M_1}(p)$, and $\text{final}_{M_2}(q)$ for all $q \in F$.*

The concept of successful branch allows for F to include multiple final states. This degree of generality supports supertypes with two or more final states (such as q_4 and q_6 of the machine N_2 of Example 1) when, later, successful branches are deployed in the context of collecting simulation graphs (see Figure 4).

Theorem 1 (Equivalence). *Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ be a collecting sim tree for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. $M_1 \leq M_2$ iff every complete branch in $(N, \xrightarrow{\ell}_t)$ is successful.*

Simulation trees and collecting simulation trees can grow without bound. However, growth can be curtailed by the judicious application of relaxation:

Definition 13 (Collecting simulation (sim) graph). *A collecting sim graph for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ is a labelled graph $(N, n_0, \xrightarrow{\ell}_g, \mathcal{L})$ where $\xrightarrow{\ell}_g \subseteq N \times N$ is a graph rooted at n_0 and $\mathcal{L} : N \rightarrow P \times \wp(T_Q)$ such that:*

1. $\mathcal{L}(n_0) = (p_0, \{q_0\})$
2. if $p \leq S \xrightarrow{\ell}_g p' \leq T$ and $\mathcal{L}(n) = (p, S)$ then there exists $n' \in N$ such that $n \xrightarrow{\ell}_g n'$, $\mathcal{L}(n') = (p', S')$ for some $S' \supseteq T$
3. if $n \xrightarrow{\ell}_g n'$ and $\mathcal{L}(n) = (p, S)$ then $\mathcal{L}(n') = (p', S')$ such that $S' \supseteq T$ and $p \leq S \xrightarrow{\ell}_g p' \leq T$

Relaxation manifests in case (2) of Definition 13 in that $S' \supseteq T$: S' is thus a relaxation of T . Note too that n' is not necessarily on the branch from n_0 to n . Case (3) ensures that each transition in a collecting sim graph has a counterpart in the collecting sim tree.

The concepts of (complete and successful) branch can be defined analogously for a collecting sim graph. With these concepts in place, the following result, which is consequence of Proposition 1, explains how a collecting sim graph simulates a collecting sim tree: each branch in the tree is described by a branch in the graph with possibly enlarged trace sets. This correspondence between a branch in the graph and a branch in the tree only holds if the branch in the collecting sim graph does not get stuck.

Corollary 1. *Let $(N, n_0, \xrightarrow{\ell}_t, \mathcal{L})$ (resp. $(N', n'_0, \xrightarrow{\ell}_g, \mathcal{L}')$) be a collecting sim tree (resp. graph) for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. If $b = n_0 \cdots n_i$ is a branch in the tree $(N, \xrightarrow{\ell}_t)$ then there exists $b' = n'_0 \cdots n'_k$ in the graph $(N', \xrightarrow{\ell}_g)$ with either: $k = i$ or $k < i$ and $n'_k \not\xrightarrow{\ell}_g$. Moreover, $\mathcal{L}(n_j) = (p_j, S_j)$, $\mathcal{L}'(n'_j) = (p_j, S'_j)$ and $S_j \subseteq S'_j$ for all $j \leq k$.*

Example 3. Figure 1 (left) shows an infinite simulation tree (following the notation of [5]) for machines M_1 and M_2 given in the introduction. The corresponding collecting sim tree has the same structure but $T_2 = \langle a : q_0 \rangle$ is substituted with $\{aq_0\}$, $T_3 = \langle a : \langle a : q_0 \rangle \rangle$ with $\{aaq_0\}$, whereas q_0 and q_1 (at and beneath the root of the tree) are replaced with $\{q_0\}$ and $\{q_1\}$ in the collecting sim tree. A (finite) collecting sim graph for M_1 and M_2 is shown in Figure 1 (right). Observe $q_0 \in S$, $q_1 \in S'$, $q_0 \in S$, $aq_0 \in S$, $aq_0 \in S'$, $aaq_0 \in S$, $aq_0 \in S$, $q_0 \in S'$, etc.

The force of collecting sim graphs is that they still act as a vehicle for establishing asynchronous subtyping, as the following result asserts:

Theorem 2 (Soundness). *Let $(N', n'_0, \xrightarrow{\ell}_g, \mathcal{L})$ be a collecting sim graph for $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$. Then $M_1 \leq M_2$ if every complete branch in $(N', \xrightarrow{\ell}_g)$ is successful.*

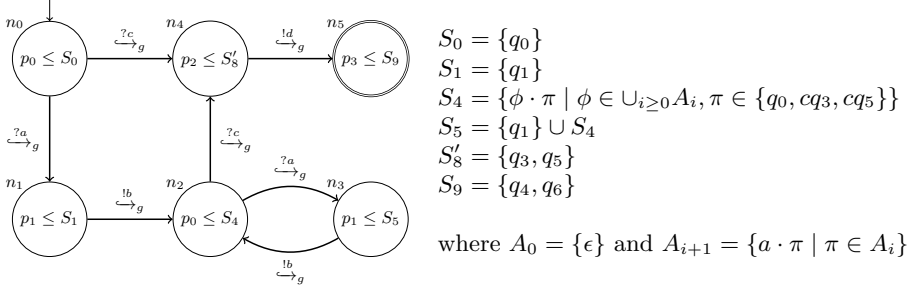


Fig. 4. A collecting sim graph for N_1 and N_2

Example 4 informally anticipates how finite representations of infinite executions can be algorithmically computed (using regular expressions) ahead of the detailed presentation and evaluation of the algorithm in the following sections.

Example 4 (Running example: collecting sim graph). Continuing with Example 1 (Figure 2), N_1 and N_2 are examples of machines for which [5] cannot prove subtyping, even though it does hold. In contrast, Figure 4 presents a collecting sim graph showing $N_1 \leq N_2$. The graph is rooted at n_0 where $\mathcal{L}(n_0) = (p_0, S_0)$.

5 Async Subtyping with Regular Expressions

Our work was motivated by the question of whether subtyping can be addressed with a simpler and more general approach. Beyond this conceptual question, there is the practical matter of whether our subtyping can algorithmically establish subtyping on more problems than before [4, 5]. To do so, we represent sets of traces using regular expressions and simulate the operations on sets of traces with analogous operations on regular expressions. To derive a finite collecting sim graph, we apply regular expression widening [12].

5.1 Representing sets of traces with regular expressions

A set of traces can be represented as a finite set of regular expressions drawn from the syntactic category Reg_A which is parameterised by alphabet A . Reg_A is inductively defined as $\text{Reg}_A = \epsilon \mid C \mid r \cdot r' \mid r^*$ where $C \subseteq A$, $r, r' \in \text{Reg}_A$, and \cdot is concatenation of words. To specify the language (set of words) represented by a regular expression, recall that Kleene closure W^* of a set of words W is defined as $W^* = \cup_{i=0}^{\infty} W_i$ where $W_0 = \{\epsilon\}$ and $W_{i+1} = \{\omega \cdot \omega' \mid \omega \in W, \omega' \in W_i\}$. Then the language of $r \in \text{Reg}_A$, denoted $\llbracket r \rrbracket$, is defined as $\llbracket \epsilon \rrbracket = \{\epsilon\}$, $\llbracket C \rrbracket = C$, $\llbracket r \cdot r' \rrbracket = \{\omega \cdot \omega' \mid \omega \in \llbracket r \rrbracket, \omega' \in \llbracket r' \rrbracket\}$ and $\llbracket r^* \rrbracket = \llbracket r \rrbracket^*$.

If $r \in \text{Reg}_A$ and $q \in Q$ the pair (r, q) represents the sets of traces $\llbracket (r, q) \rrbracket = \{\pi \cdot q \mid \pi \in \llbracket r \rrbracket\}$. Furthermore, if $R \subseteq \text{Reg}_A \times Q$ then R represents the traces $\llbracket R \rrbracket = \cup \{\llbracket (r, q) \rrbracket \mid (r, q) \in R\}$. Henceforth rq will abbreviate the pair (r, q) .

Example 5. To illustrate, $\llbracket \{a^*q_0, cq_3\} \rrbracket = \{cq_3\} \cup \{\pi \cdot q_0 \mid \pi \in \cup_{i \geq 0} A_i\}$ with A_i defined as in Figure 4.

Our technique uses the existing notion of widening [15, 16] to approximate regular expressions, namely to relax a sequence of regular expressions to derive another sequence which is not strictly increasing (thereby inducing convergence):

Definition 14. An operation $\nabla : \text{Reg}_A \times \text{Reg}_A \rightarrow \text{Reg}_A$ is a widening iff given a sequence $s_0, s_1, \dots \in \text{Reg}_A$ such that $\llbracket s_i \rrbracket \subseteq \llbracket s_{i+1} \rrbracket$ for all $i \geq 0$, the (widened) sequence $w_0 = s_0$ and $w_{i+1} = w_i \nabla s_{i+1}$ satisfies the following properties:

- $\llbracket s_i \rrbracket \subseteq \llbracket w_i \rrbracket$ and $\llbracket w_i \rrbracket \subseteq \llbracket w_{i+1} \rrbracket$ for all $i \geq 0$
- the sequence $\llbracket w_0 \rrbracket, \llbracket w_1 \rrbracket, \dots$ is not strictly increasing

Our approach is parametric on the widening (of which there are many [14]). We provide a primer on (string) widening to keep the presentation self-contained.

5.2 Widening regular expressions (a self-contained primer)

The intuition behind the widening we adopt [12] is to preserve commonality across two regular expressions and resolve any difference using Kleene star for relaxation. The widening scans both expressions left-to-right and, as it does so, it partitions each expression into a prefix p which has been traversed and a suffix s which is yet to be considered. The state of the scan thus represented by a pair (p, s) , with widen_k operating on two such pairs simultaneously:

$$\begin{aligned} \text{widen}_k((p, \epsilon), (p', s')) &= \text{mash}_k(p, p' \cdot s') & \text{widen}_k((p, s), (p', \epsilon)) &= \text{mash}_k(p \cdot s, p') \\ \text{widen}_k((p, q \cdot s), (p', q' \cdot s')) &= \\ &\begin{cases} \text{mash}_k(p, p') \circ q \circ \text{widen}_k((\epsilon, s), (\epsilon, s')) & \text{if } q = q' \text{ and } \text{sh}(q) \leq k \\ \text{widen}_k((p \cdot q, s), (p' \cdot q, s')) & \text{if } q = q' \text{ and } \text{sh}(q) > k \\ \text{widen}_k((p \cdot q, s), (p', q' \cdot s')) & \text{if } q \neq q' \text{ and } |s| > |s'| \\ \text{widen}_k((p, q \cdot s), (p', q' \cdot s')) & \text{if } q \neq q' \text{ and } |s| \leq |s'| \end{cases} \end{aligned}$$

The widening is defined in terms of two notions of size: (1) *star height* defined $\text{sh}(\epsilon) = \text{sh}(C) = 0$, $\text{sh}(r^*) = \text{sh}(r) + 1$ and $\text{sh}(r \cdot s) = \max(\text{sh}(r), \text{sh}(s))$; (2) *star length* defined $|\epsilon| = 0$, $|C| = |r^*| = 1$ and $|r \cdot s| = |r| + |s|$. Given two expressions r and s , the auxiliary $\text{mash}_k(r, s)$ computes a relaxation of r and s such that $\text{sh}(\text{mash}_k(r, s)) \leq k$ where k is a predefined depth bound. Thus $\llbracket r \rrbracket \subseteq \llbracket \text{mash}_k(r, s) \rrbracket$ and $\llbracket s \rrbracket \subseteq \llbracket \text{mash}_k(r, s) \rrbracket$.

Now consider scans of the form $(p, q \cdot s)$ and $(p', q' \cdot s')$ where q and q' are sub-expressions of the form C or r^* . If $q = q'$ then the common q is preserved provided $\text{sh}(q) \leq k$ and widening continues with scans (ϵ, s) and (ϵ, s) . Operator \circ is concatenation followed by a normalisation step [12] which ensures that no consecutive stars are introduced. If $\text{sh}(q) > k$ both q and q' are appended onto r and r' to be relaxed subsequently by mash_k . If $q \neq q'$ either q or q' is appended onto its prefix depending on $|s| > |s'|$ so that the remaining suffices are closer in length (which is merely a heuristic for improving their similarity). Analogous to mash_k , $\text{widen}_k((p, s), (p', s'))$ relaxes $p \cdot s$ and $p' \cdot s'$ such that $\text{sh}(\text{widen}_k((p, s), (p', s')))) \leq k$. The star height bound ensures $r \nabla s = \text{widen}_k((\epsilon, r), (\epsilon, s))$ yields a sequence which is not strictly increasing [12].

Algorithm 1 Algorithm for async subtyping ($\xrightarrow{\ell}$ is defined in Figure 3)

```

1: function SUBTYPE( $M_1, M_2, \Delta$ ) //  $M_1 = (P, p_0, \delta_1)$   $M_2 = (Q, q_0, \delta_2)$ 
2:   for ( $p \in P$ ) do
3:     if ( $\Delta(p) \neq \emptyset \wedge p \leq \Delta(p) \not\rightarrow$ ) then return maybe
4:      $R_p := \bigcup_{p' \in P} \{R \mid \exists \ell. p' \leq \Delta(p') \xrightarrow{\ell} p \leq R\}$ 
5:      $\Delta'(p) :=$  if ( $p \in wp$ ) then  $\Delta(p) \nabla R_p$  else  $\Delta(p) \cup R_p$ 
6:   if ( $\Delta' \subseteq \Delta$ ) then return  $\Delta$ 
7:   return SUBTYPE( $M_1, M_2, \Delta'$ )

```

Example 6. For brevity, we refer the reader to [12] for a definition and commentary on the auxiliary $\text{mash}_k(r, s)$ but note that $\text{mash}_k(r, \epsilon) = r^*$ if $\text{sh}(r^*) \leq k$ and conversely $\text{mash}_k(\epsilon, s) = s^*$ if $\text{sh}(s^*) \leq k$. Hence

$$\begin{aligned}
(a \cdot c \cdot d) \nabla (a \cdot b \cdot c) &= \text{widen}_1((\epsilon, a \cdot c \cdot d), (\epsilon, a \cdot b \cdot c)) = \epsilon \cdot a \cdot \text{widen}_1((\epsilon, c \cdot d), (\epsilon, b \cdot c)) \\
&= \epsilon \cdot a \cdot \text{mash}_1(\epsilon, b) \cdot c \cdot \text{widen}_1((\epsilon, d), (\epsilon, \epsilon)) \\
&= \epsilon \cdot a \cdot b^* \cdot c \cdot \text{mash}_1(d, \epsilon) = \epsilon \cdot a \cdot b^* \cdot c \cdot d^*
\end{aligned}$$

The widening can be lifted from a pair of regular expressions to a pair of sets of regular expressions in a point-wise fashion [12]. In our setting, regular expressions represent traces, where each trace takes the form rq , and thus it is natural to partition a set of traces according to the state q in which they end. Two sets of expressions can be widened point-wise, for each q separately.

5.3 Computing a collecting sim graph with regular expressions

Before outlining the algorithm, we illustrate it by example. Example 7 revisits Example 4 and shows how the sets of traces in Figure 2 can be algorithmically generated by using regular expressions and widening in tandem.

Example 7. Figure 5 presents a collecting sim graph for $N_1 \leq N_2$. Some nodes are shadowed by grey nodes that elaborate their relaxations by widening or union. The construction of the graph commences at node for $p_0 \leq R_0$ and proceeds iteratively, the number to the top-right of a node indicating the iteration at which that node is added to the graph. Iteration 1 is computed merely using the rules of Figure 3. On iteration 2, $p_0 \leq R_2$ is computed, again using the rules. Since p_0 was visited before, to ensure that p_0 is not revisited ad infinitum, a relaxation is applied, denoted ∇ following [15, 16], which relaxes R_2 using R_0 to obtain R'_2 . Observe how $\llbracket R_0 \rrbracket \subseteq \llbracket R'_2 \rrbracket$ and $\llbracket R_2 \rrbracket \subseteq \llbracket R'_2 \rrbracket$ but crucially the regular expression R'_2 is computed using a (widening) algorithm [12] which ensures that only a finite number of regular expressions are ever generated for p_0 . Not all nodes of Figure 5 need to be relaxed using widening. On iteration 3, p_1 is revisited. In this case, R'_3 is derived from R_3 and R_1 by computing their union. Thus again $\llbracket R_1 \rrbracket \subseteq \llbracket R'_3 \rrbracket$ and $\llbracket R_3 \rrbracket \subseteq \llbracket R'_3 \rrbracket$. The general strategy is to apply widening only as required, namely on a set of nodes which cut any cycle [3]. The machine N_1 of Figure 2 has a single cycle through p_0 and p_1 , thus it is sufficient to widen

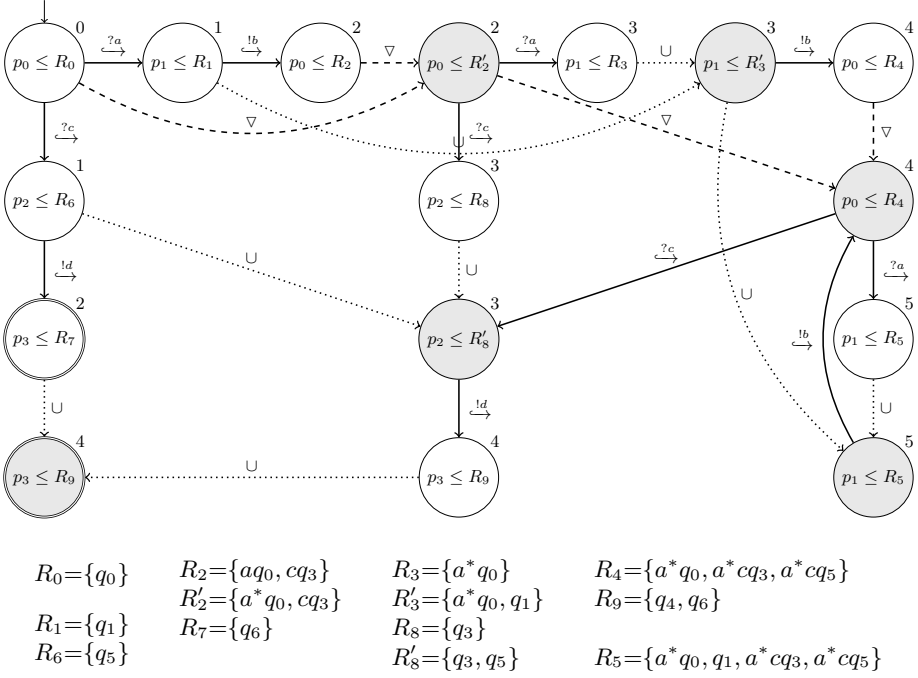


Fig. 5. A collecting simulation graph for proving $N_1 \leq N_2$: reprise

at either p_0 or p_1 . We elect to widen at p_0 , whereas for all other nodes of N_2 , the relaxation is union. On iteration 5, $p_1 \leq R_5$ is computing as before, the union of R'_3 with R_5 being R_5 . The following $\xrightarrow{?b}$ transition derives a regular expression R which is subsumed by R_4 , that is, $p_1 \leq R_5 \xrightarrow{?b} p_0 \leq R$ where $\llbracket R \rrbracket \subseteq \llbracket R_4 \rrbracket$. Thus the graph is no longer developed along the cycle. Despite employing relaxation, R_9 only contains q_4 and q_6 for which $\text{final}_{N_2}(q_4)$ and $\text{final}_{N_1}(q_6)$ hold. Recall $\text{final}_{N_1}(p_3)$ holds, hence subtyping is demonstrated.

Our SUBTYPE algorithm takes as input two machines $M_1 = (P, p_0, \delta_1)$ and $M_2 = (Q, q_0, \delta_2)$ and is parametric on: (1) a widening $\nabla : \wp(\text{Reg}_A) \times \wp(\text{Reg}_A) \rightarrow \wp(\text{Reg}_A)$ and (2) a set $wp \subseteq P$ of widening points. At least one state of wp must appear in any cycle of M_1 ; a condition which is sufficient for widening to induce termination [3]. The mapping $\Delta : P \rightarrow \wp(\text{Reg}_A \times Q)$ represents the nodes of an evolving collecting sim graph: SUBTYPE(M_1, M_2, Δ) is initially primed with $\Delta = \lambda p. \text{if } (p = p_0) \text{ then } \{(\epsilon, q_0)\} \text{ else } \emptyset$. In line 3, **maybe** is returned if the simulation gets stuck. Note that $p \leq R \xrightarrow{\ell} p' \leq R'$ abbreviates $p \leq \llbracket R \rrbracket \xrightarrow{\ell} p' \leq \llbracket R' \rrbracket$ and likewise $p \leq R \not\xrightarrow{\ell} p' \leq R'$ abbreviates $p \leq \llbracket R \rrbracket \not\xrightarrow{\ell} p' \leq \llbracket R' \rrbracket$. In line 4, R_p collects all the (r, q) pairs reachable at p in the current iteration Δ . $\Delta(p)$ is then relaxed to $\Delta'(p)$ applying widening if $p \in wp$ and union otherwise. In line 5, $\Delta' \subseteq \Delta$ iff $\llbracket \Delta(p) \rrbracket \subseteq \llbracket \Delta'(p) \rrbracket$ for all $p \in P$. This check determines whether a fix-point is reached: if so the

M_1		M_2		$ M_1 $	$ M_2 $	[5]	regex	time
ct ta1	ct ta2	7	5	✗	✓	110		
ct tb1	ct tb2	6	7	✗	✓	41		
14may2	14may1	4	7	✗	✓	10		
badseq1	badseq2	5	12	✗	✓	1127		
march3testa1	march3testa2	6	7	✗	✓	222		
aaaaaab1	aaaaaab2	5	3	✗	✓	43		
e 1okloop	e 2okloop	10	8	✗	✓	1757		
march3testa1	march3testb2	6	10	✗	✗	8		

Fig. 6. Comparison of subtyping experiments: success rates and execution time (in ms)

algorithm returns Δ . SUBTYPING is sound and, due to widening, is guaranteed to terminate. In short, if SUBTYPING returns Δ then $M_1 \leq M_2$, otherwise it returns *maybe* and the subtyping check is deemed inconclusive.

For complexity, observe that wp can be chosen so that each state of $P \setminus wp$ has at most one incoming edge. Then algorithm 1 updates each state of P at most $(c|Q|)^{|wp|}$ times, updating Δ at most $|P|(c|Q|)^{|wp|}$ times, where c bounds the number of times a regular string can be relaxed. But $c \leq (2|Q|)^{k+2}$.

5.4 Implementation and benchmarking

If successful, our tool generates a collecting sim graph (in the form of Δ) which provides a concrete artifact that certifies subtyping. The regular expression-based subtyping algorithm has been implemented in Scala 3.2.2 on a laptop running Ubuntu 22.04.2 with 32 GB of DDR3 and a 2.8GHz Intel i7 processor. The code base is 1059 LOC, making use of parser combinators and the mutable and immutable Set libraries. No attempt has been made to improve the iteration strategy (which is normally a source of speedups). The tool and benchmarks are available at <https://github.com/murgia88/AsynchSubtypingRegex>. The benchmarks³ consists of 175 pairs of session types: 83 pairs where one type is known to be a subtype of the other (the positive problems); and 92 pairs which are known not to be in a subtyping relation (the negative problems).

This is a positive outcome. Alternatively, the algorithm terminates with an inconclusive verdict. We have applied our tool to all the subtyping problems in the benchmarking suite. Our tool gave positive outcome for 82 of them, whereas the tool in [4, 5] gave 75 positive outcomes. In addition to certifying all positive cases in [4, 5], the tool could certify 7 “complex accumulation [input tree] patterns” [5] that were inconclusive cases in previous work. All 92 negative problems were (rightly) categorised as inconclusive by our tool.

An analysis of the 7 complex accumulation patterns is summarised in Figure 6. The M_1 (resp. M_2) column give the candidate subtype (resp. type). To

³ The suite is based on the benchmark in [4, 5] with the addition of one (positive) case that is used in [4, 5] as a running example.

convey some indication of the size of the problems, the $|M_1|$ (resp. $|M_2|$) column gives the number of states in M_1 (resp. M_2). The [5] column indicates whether subtyping can be proven using the algorithm of [5] using their distribution. The regex column indicates whether subtyping can be proven using collecting sim graphs instantiated with regular expressions, as proposed in our work. Time is walltime measured in milliseconds, the median of 5 runs. Widening was performed with a maximum star height of just 1 ($k = 1$). The last example in Figure 6, `marchtesta1` \leq `marchtestb2`, is known to be positive but neither our tool nor the one in [4, 5] could prove it. Nevertheless, it is remarkable that the widening of [12] performs so well considering it was originally devised for extracting SQL queries from database application programs.

The certificate produced by the algorithm (in the form of Δ) can be checked against the rules of Figure 3, without using widening or iteration. This could conceivably be performed by a proof assistant for high-assurance applications.

We finally comment on one complex example, `marchtesta1` \leq `marchtestb2`, that neither our tool nor the one in [4, 5] could prove. A post mortem reveals that $p_4 \leq S_4$ gets stuck: traces of S_4 of the form $b\pi q_3$ cannot make any move thus `RecvSet` does not apply. However, $b\pi q_3$ originates from $\{a, b\}^*q_3$ in $p_0 \leq S_0$ which itself stems from $(\epsilon q_3 \nabla_1 a q_3) \nabla_1 b(\epsilon q_3 \nabla_1 a q_3)$. Setting $k = 2$ (or higher) does not remedy the problem, which suggests that the widening needs tuning. Indeed, replacing $\{a, b\}^*q_3$ in S_0 with a more nuanced relaxation, namely $(a^*(ba)^*a^*)^*q_3$, is sufficient to establish subtyping. Crucially, this shows that the problem does not lie in collecting sim graph construction itself but in the widening (something which can be tuned without change to the underlying framework).

6 Conclusion and Related Work

We presented an algorithm for (binary) asynchronous session subtyping based on the application of abstract interpretation to session types. Our approach centres on the use of sets of traces to obtain a tractable representation of input trees. Sets of traces allow us to separate the proof for correctness of the core algorithm, from the problem of how to finitely represent and manipulate traces. This separation makes the methodology modular and tunable. As well as providing a conceptually simple approach for proving subtyping, the resulting algorithm, when instantiated with an off-the-shelf string widening, can prove subtyping for rich forms of interaction that were previously out-of-reach [5]. From a large suite of benchmarks, our algorithm was able to verify subtyping all but one problem and, even for that, we have shown that the collecting simulation approach is still adequate for proving subtyping. These results show that abstract interpretation is a clean, useful and powerful vehicle for inferring subtyping. Furthermore, a collecting sim graph once obtained constitutes a *certificate* for validating subtyping. The certificate can be then checked by a third-party, without consideration for how the graph is actually derived (whether algorithmically or manually).

Related work Async subtyping was first explored in [29] where subtyping rules consider a restricted form of permutation on actions. These concepts were then

refined [10, 11] to disallow orphan messages, a requirement adopted in [5] and inherited into our study for ease of comparison.

Since async subtyping is undecidable [6, 27], some works proposed decidable safe approximated algorithms. For instance, subtyping can be approximated by k -bounded asynchronous subtyping [7]. The state of the art is [4, 5] that inspired our work. Fragments of session types for which asyn subtyping is decidable include: alternating session types [7] and single-out (resp. single-in) types [7] where internal (resp. external) choices are singletons.

Fair subtyping [9, 33] is an alternative to standard subtyping that preserves the possibility of correct termination. Asynchronous fair subtyping [8] is undecidable, and a sound algorithm has been proposed [8], which extends [5]. We would expect trace relaxation to extend to this setting as well.

The work above mostly focuses on binary sessions. The subtyping algorithm of [17], instead, focuses on the more general case of async *multiparty* subtyping. When restricted to binary types, the algorithm in [17] is less powerful than both [5] and our algorithm. The last case of [17, Table 1], taken from the running example in [5], is undetected with deadlock-free subtyping [17] but is proven by [5] and ourselves (see case ‘ $\text{sub} - \text{runningex} \leq \text{sup} - \text{runningex}$ ’ in <https://github.com/murgia88/AsynchSubtypingRegex>). [17] is still able to establish subtyping for several realistic protocols. A precise definition of async multiparty subtyping (AMS) has been provided in Ghilezan et al. [22]. This means that AMS in [22] is sound and complete with respect to async multiparty typing with a subsumption rule. Such definition is not obviously useful for algorithmic purposes: it contains quantifications over uncountably infinite sets. Application of our methodology to AMS is an interesting future direction.

Acknowledgements This work has been partially supported by EPSRC project EP/T014512/1 (STARDUST), the BehAPI project funded by the EU H2020 RISE under the Marie Skłodowska-Curie action (No: 778233) and MUR project PON REACT EU DM 1062/21.

References

1. Bartoletti, M., Murgia, M., Scalas, A., Zunino, R.: Verifiable Abstractions for Contract-oriented Systems. *J. Log. Algebraic Methods Program.* **86**(1), 159–207 (2017)
2. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring Networks through Multiparty Session Types. *Theoretical Computer Science* **669**, 33–58 (2017), <https://doi.org/10.1016/j.tcs.2017.02.009>
3. Bourdoncle, F.: Efficient Chaotic Iteration Strategies with Widenings. In: *Formal Methods in Programming and Their Applications*. Lecture Notes in Computer Science, vol. 735, pp. 128–141. Springer-Verlag (1993). <https://doi.org/10.1007/BFb0039704>
4. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A Sound Algorithm for Asynchronous Session Subtyping. In: *International Conference on Concurrency Theory*. *LIPICs*, vol. 140, pp. 38:1–38:16. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2019), <http://dx.doi.org/10.4230/LIPICs.CONCUR.2019.38>

5. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *Logical Methods in Computer Science* **17**(1), 1–35 (2021). [https://doi.org/10.23638/LMCS-17\(1:20\)2021](https://doi.org/10.23638/LMCS-17(1:20)2021)
6. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of Asynchronous Session Subtyping. *Information and Computation* **256**, 300–320 (2017), <https://doi.org/10.1016/j.ic.2017.07.010>
7. Bravetti, M., Carbone, M., Zavattaro, G.: On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping. *Theoretical Computer Science* **722**, 19–51 (2018), <https://doi.org/10.1016/j.tcs.2018.02.010>
8. Bravetti, M., Lange, J., Zavattaro, G.: Fair Refinement for Asynchronous Session Types. In: *Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science*, vol. 12650, pp. 144–163. Springer-Verlag (2021). https://doi.org/10.1007/978-3-030-71995-1_8
9. Bravetti, M., Zavattaro, G.: A Foundational Theory of Contracts for Multi-party Service Composition. *Fundamenta Informaticae* **89**(4), 451–478 (2008)
10. Chen, T.C., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science* **13**(2), 1–61 (2017). [https://doi.org/10.23638/LMCS-13\(2:12\)2017](https://doi.org/10.23638/LMCS-13(2:12)2017)
11. Chen, T.C., Dezani-Ciancaglini, M., Yoshida, N.: On the Preciseness of Subtyping in Session Types. In: *Principles and Practice of Declarative Programming*. pp. 135–146. ACM Press (2014). <https://doi.org/10.1145/2643135.2643138>
12. Choi, T., Lee, O., Kim, H., Doh, K.: A Practical String Analyzer by the Widening Approach. In: *Asian Symposium on Programming and Systems. Lecture Notes in Computer Science*, vol. 4279, pp. 374–388. Springer-Verlag (2006). https://doi.org/10.1007/11924661_23
13. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: *Static Analysis Symposium. Lecture Notes in Computer Science*, vol. 2694, pp. 1–18. Springer-Verlag (2003). https://doi.org/10.1007/3-540-44898-5_1
14. Costantini, G., Ferrara, P., Cortesi, A.: A Suite of Abstract Domains for Static Analysis of String Values. *Software Practice and Experience* **45**, 245–287 (2015)
15. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Principles of Programming Languages*. pp. 238–252. ACM Press (1977), <https://doi.org/10.1145/512950.512973>
16. Cousot, P., Cousot, R.: Comparing the Galois connection and Widening/Narrowing approaches to Abstract Interpretation. In: *Programming Language Implementation and Logic Programming*. pp. 269–295. No. 631 in *Lecture Notes in Computer Science*, Springer-Verlag (1992), https://doi.org/10.1007/3-540-55844-6_142
17. Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In: *Symposium on Principles and Practice of Parallel Programming*. pp. 246–261. ACM Press (2022). <https://doi.org/10.1145/3503221.3508404>
18. Demangeon, R., Honda, K.: Full Abstraction in a Subtyped pi-Calculus with Linear Types. In: *International Conference on Concurrency Theory. Lecture Notes in Computer Science*, vol. 6901, pp. 280–296. Springer-Verlag (2011). https://doi.org/10.1007/978-3-642-23217-6_19
19. Deniérou, P.M., Yoshida, N.: Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In: *International Colloquium on Automata, Languages and Programming. Lecture*

- Notes in Computer Science, vol. 7966, pp. 174–186. Springer-Verlag (2013). https://doi.org/10.1007/978-3-642-39212-2_18
20. Gay, S., Hole, M.: Types and Subtypes for Client-Server Interactions. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 74–90. Springer-Verlag (1999)
 21. Gay, S., Hole, M.: Subtyping for Session Types in the Pi Calculus. *Acta Informatica* **42**, 191–225 (2005). <https://doi.org/10.1007/s00236-005-0177-z>
 22. Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., Yoshida, N.: Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.* **5**(POPL), 1–28 (2021). <https://doi.org/10.1145/3434297>
 23. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 1381, pp. 22–138. Springer-Verlag (1998). <https://doi.org/10.1007/BFb0053567>
 24. Hu, R., Yoshida, N.: Hybrid Session Verification Through Endpoint API Generation. In: Formal Aspects of Software Engineering. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer-Verlag (2016). https://doi.org/10.1007/978-3-662-49665-7_24
 25. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 5142, pp. 516–541. Springer-Verlag (2008). https://doi.org/10.1007/978-3-540-70592-5_22
 26. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In: European Conference on Object-Oriented Programming. vol. 222, pp. 4:1–4:29. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ECOOP.2022.4>
 27. Lange, J., Yoshida, N.: On the Undecidability of Asynchronous Session Subtyping. In: Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 10203, pp. 441–457. Springer-Verlag (2017), https://link.springer.com/chapter/10.1007/978-3-662-54458-7_26
 28. Lindley, S., Morris, J.G.: Embedding session types in Haskell. In: International Symposium on Haskell. pp. 133–145. ACM Press (2016). <https://doi.org/10.1145/2976002.2976018>
 29. Mostrous, D., Yoshida, N., Honda, K.: Global Principal Typing in Partially Commutative Asynchronous Sessions. In: European Symposium on Programming. Lecture Notes in Computer Science, vol. 5502, pp. 316–332. Springer-Verlag (2009). https://doi.org/10.1007/978-3-642-00590-9_23
 30. Neykova, R., Hu, R., Yoshida, N., Abdeljalal, F.: A Session Type Provider: Compile-time API Generation of Distributed Protocols with Interaction Refinements in F#. In: Compiler Construction. pp. 128–138. ACM Press (2018). <https://doi.org/10.1145/3178372.3179495>
 31. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe Parallel Programming with Message Optimisation. In: Objects, Models, Components, Patterns. Lecture Notes in Computer Science, vol. 7304, pp. 202–218. Springer-Verlag (2012). https://doi.org/https://doi.org/10.1007/978-3-642-30561-0_15
 32. Orchard, D., Yoshida, N.: Session Types with Linearity in Haskell. In: Behavioural Types: from Theory to Tools. pp. 219–241. River Publishers (2017). <https://doi.org/https://doi.org/10.13052/rp-9788793519817>
 33. Padovani, L.: Fair subtyping for multi-party session types. In: Coordination Models and Languages. Lecture Notes in Computer Science, vol. 6721, pp. 127–141. Springer-Verlag (2011). https://doi.org/10.1007/978-3-642-21464-6_9

34. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Parallel Architectures and Languages Europe. Lecture Notes in Computer Science, vol. 817, pp. 398–413. Springer-Verlag (1994). https://doi.org/10.1007/3-540-58184-7_118

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.



Program Analysis and Proofs



SootUp: A Redesign of the Soot Static Analysis Framework

Kadiray Karakaya¹, Stefan Schott¹, Jonas Klauke¹, Eric Bodden^{1,2},
Markus Schmidt¹, Linghui Luo^{3(✉)}, and Dongjie He⁴

¹ Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany

{kadiray.karakaya, stefan.schott, jonas.klauke,
eric.bodden, markus.schmidt}@upb.de

² Fraunhofer IEM, Paderborn, Germany

³ Amazon Web Services, Berlin, Germany
llinghui@amazon.de

⁴ Chongqing University, Chongqing, China
dongjiehe@cqu.edu.cn

Abstract. Since its inception two decades ago, SOOT has become one of the most widely used open-source static analysis frameworks. Over time it has been extended with the contributions of countless researchers. Yet, at the same time, the requirements for SOOT have changed over the years and become increasingly at odds with some of the major design decisions that underlie it. In this work, we thus present SOOTUp, a complete reimplementation of SOOT that seeks to fulfill these requirements with a novel design, while at the same time keeping elements that SOOT users have grown accustomed to.

Keywords: Static program analysis · Soot · SootUp.

1 Introduction

SOOT is a program analysis framework for Java and Android. It has been popular in academia for prototyping novel static and dynamic analysis approaches, many of which have been published at international conferences [1, 3, 5, 6, 14, 15, 20, 21, 23, 29]. In 2000 [30], SOOT was introduced as an optimization framework for Java. Back then, when just-in-time compilers were still in their infancy, ahead-of-time optimization of Java code was a major field of research. Over the years, the research community’s interest has been dominantly shifting to static code analysis, for diverse purposes. SOOT remained relevant due to some of its strengths, particularly its popular intermediate representations.

One of the core features of SOOT is its main intermediate representation (IR), JIMPLE [31]. When seeking to perform program analysis on Java, both bytecode and source code are usually suboptimal representations to work with. Java bytecode represents a program to be *executed*, using a stack-based instruction set. Java source code, on the other hand, represents it on a higher level, using

Linghui Luo: The work was done prior to joining Amazon.

nested scopes and control-flow constructs for better *readability*. Soot’s JIMPLE IR is a so-called three-address code representation [13] that combines the best of both worlds: It uses local variables instead of a stack. This simplifies data-flow equations because all values that an operation consumes or produces are readily accessible through its operands. It also uses explicit control flow without nesting, i.e., solely through conditional or unconditional gotos. In result, every JIMPLE instruction is atomic, there can be no nesting. Complex source-code statements, which perform multiple consecutive operations, e.g. a numerical computation with a subsequent cast, are broken down into multiple individual IR instructions. This enables the creation of simple control flow graphs (CFGs), which one can then use to analyze a method’s control and data flow with relative ease.

Furthermore, SOOT offers multiple algorithms, with varying degrees of precision and complexity, for constructing call graphs. They resemble an essential data structure for performing inter-procedural static analysis, as it models how a program’s methods call one another. For object-oriented programming languages like Java, call graph construction is particularly challenging. This is because in Java method calls are virtual by default, in which case their call target is dependent on an object’s runtime type. A reference variable’s declared type can only bound the possible call targets. To resolve call targets precisely one must compute all of the variable’s possible runtime types. A popular way to do this is through pointer analysis. SOOT provides such call graph computation through its pointer analysis framework SPARK.

Over the years, SOOT has frequently been extended to incorporate new features, and, in doing so, even early on it became clear that some of its design decisions were suboptimal, yet hard to remedy after the fact. For instance, SOOT has always been all-round monolithic. It heavily uses the singleton design pattern, causing strong coupling, and it always sought to be both a command line tool and a library, causing sometimes conflicting views on who owns the thread of control. In SOOT, everything can be accessed and manipulated via the *singleton* “scene”. This forbids keeping multiple scenes in memory, and any sensible parallelization. SOOT also contains many features that by now are considered obsolete, e.g. other barely used IRs and an outdated source-code frontend, which are hard to remove without breaking useful but *untested* functionality.

This paper presents SOOT’s successor framework SOOTUP. With SOOTUP, we aim to keep the most important features of SOOT, yet to also overcome its major drawbacks. We designed SOOTUP as a *modular library*. This allows one to pick out the necessary modules for a specific use case. For instance, clients that only require bytecode analysis would add a dependency to the bytecode frontend module. This is possible due to SOOTUP’s core module being a generic implementation that allows plugging in frontends for arbitrary programming languages. Instead of a singleton scene object, SOOTUP introduces the concept of *views*, where each view may hold a different version of the analyzed program, or different programs altogether. To enable safe parallelization and caching, the new JIMPLE IR is immutable by default, allowing instrumentation only at certain

safe points. At the time of writing, SOOTUP’s most recent release is v1.1.2¹ and SOOTUP is open-sourced at GitHub.²

To summarize, this paper presents the following contributions:

- The design decisions behind SOOTUP’s architecture that accommodate current research requirements,
- a demonstration of its new API, which aims for better usability,
- suggestions for SOOT-based analysis tools on how to switch to SOOTUP, and
- the roadmap for further development of SOOTUP.

The remainder of this paper is organized as follows. In Section 2, we introduce the design decisions that shaped SOOTUP. In Section 3, we demonstrate the new API on example use cases. In Section 4, we list currently supported tools and discuss how to upgrade tools to use SOOTUP. In Section 5, we explain SOOTUP’s development process and how one can contribute to it. We present the future work in Section 6, related work in Section 7 and conclude with Section 8.

2 Design Decisions

We next discuss the main design decisions that underly SOOTUP, and how they address some of the major shortcomings of SOOT. We introduce the new architecture and excerpts of the new API.

2.1 Modular Architecture

SOOTUP’s most notable architectural difference from its predecessor is the clear separation of its components into independent modules. Figure 1 shows its architectural overview. One of the goals of the new architecture is to allow SOOTUP to be used as a language-*independent* static analysis framework. It is not tightly coupled to any programming language. The most recent release (v1.1.2) includes frontends for Java bytecode, Java source code and a now generic, i.e., language-independent form of JIMPLE. We delegate the language support to external frontend providers and expect them to *extend* the *generic* JIMPLE. This is a significantly different mechanism than SOOT had offered for language support before. Previously, to analyze programs not in Java, one needed to convert their code to the (*Java-specific*) JIMPLE. With SOOTUP, instead one defines language-specific features by extending the core set of JIMPLE language constructs.

The *core* module encapsulates the main functionality based on the generic JIMPLE. It defines the JIMPLE language constructs such as expressions, constants and statements. The statements make up control-flow graphs (CFGs), which may be forward, backward, mutable or immutable. The CFGs are representations for the bodies of `SootMethods`. `SootMethods` constitute `SootClasses`, the backbone of SOOTUP’s *core object model*. All of these objects are accessible through `Views`.

¹ <https://doi.org/10.5281/zenodo.10037587>

² <https://github.com/soot-oss/SootUp/>

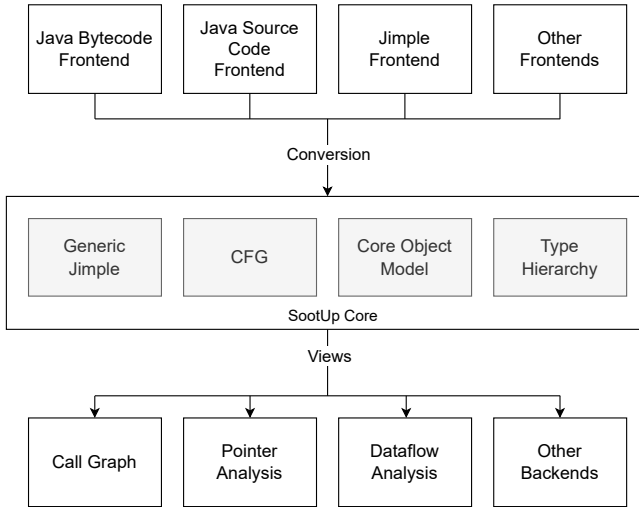


Fig. 1. Overview of SOOTUP’s Architecture. White boxes are Java modules.

We have conceptualized the **View** as the main interface the user interacts with. In the case of a single view, this corresponds to the **Scene** object in SOOT. Because of the **Scene**’s singleton nature, running multiple analyses simultaneously was virtually impossible in SOOT [16]. SOOTUP overcomes this drawback by allowing as many **Views** as desired to co-exist.

Additionally, SOOTUP comes with a new extensible *Call Graph* framework. It allows plugging in arbitrary strategies for resolving virtual method dispatches. These strategies could vary, for instance, to optimize the precision or scalability, which are often tweaked using different *Pointer Analysis* algorithms. Interprocedural *Dataflow Analysis* is one of the most successful methods for detecting bugs and security vulnerabilities. SOOTUP supports out-of-the-box context-sensitive data-flow analysis using the popular HEROS [4] dataflow analysis framework.

2.2 On-Demand Class Loading

While SOOT loads all **SootClasses** that are referenced in a currently resolving **SootClass**, SOOTUP is designed with a layer of indirection. SOOTUP makes use of identifiers to reference actual, possibly already loaded, instances of a respective **SootClass** and stores those identifiers that reference other **SootClasses**, **SootMethods** or **SootFields**. This decreases unnecessary computations of unused **SootClasses**, i.e. those which are referenced but whose contents are not of interest. Doing so, additionally, enables parallel class loading. Because the loading of a class does not depend on the loading of the classes that it references, each class can be loaded independently. As a side effect, it renders the concept of *phantom classes*, known from Soot, obsolete, as its purpose is to create a facade **SootClass** in case of missing a class definition of a referenced **SootClass**.

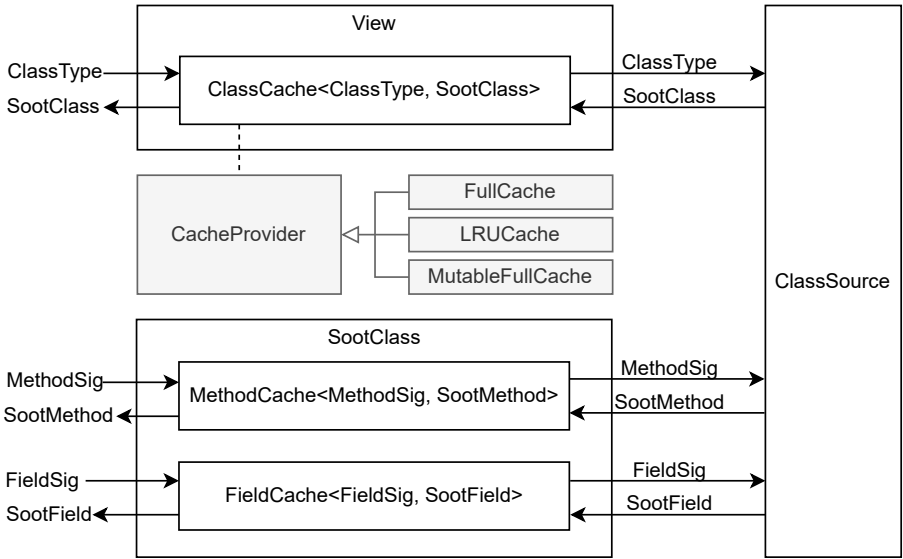


Fig. 2. SOOTUP's On-Demand Class Loading Mechanism

This case is now cleanly handled by the **View**, which simply returns no further information.

Figure 2 models SOOTUP's new on-demand class loading mechanism. The **View** is the central access point that streamlines the resolving and caching process. The caching strategy can be configured by using one of the cache providers. **FullCache** is the default option, which suffices in most cases where the cache does not need to be freed. Alternatively **LRUCache** manages the cache based on the least recent use and **MutableFullCache** gives the control of the cache to the client. After obtaining a **SootClass**, by querying it with its unique identifier (**ClassType**) from the **View**, one can obtain its **SootMethods** and **SootFields** that are cached within the **SootClass**.

2.3 Focus on an Intuitive API

SOOT's users often complain about a lack of documentation. Its issue tracker is filled with "how to"³ questions. We believe the underlying problem is, primarily, its complicated API design. Based on our past experience, when developing SOOTUP, an intuitive API design has always been strongly in focus.

Figure 3 shows the process of setting up a **Project**, creating a **View** and accessing a **SootMethod** object. First, users create an **AnalysisInputLocation** that points to a target program's path. Second, they create a **Project** by specifying the target language. The **Project** can be used to create a **View**. At this

³ <https://github.com/soot-oss/soot/issues?q=how+to+in%3Atitle>

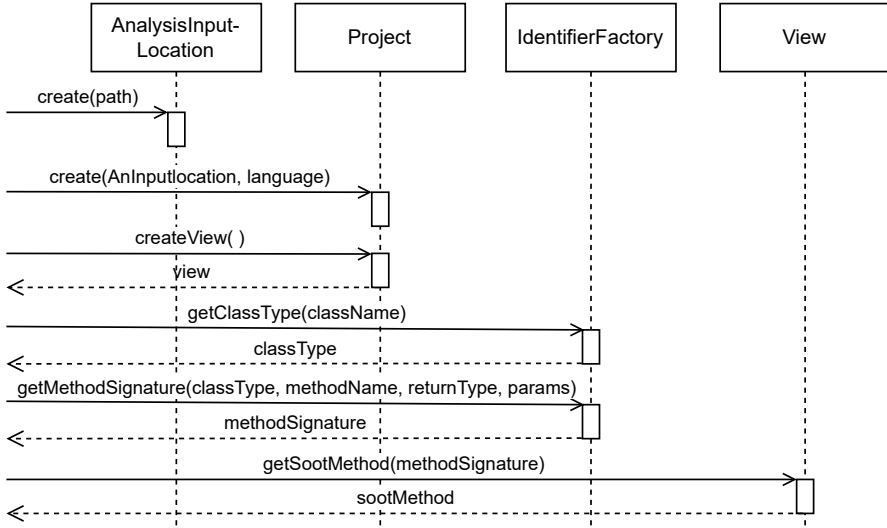


Fig. 3. SOOTUP’s API for Creating a View and Accessing a SootMethod

point, the **View** knows where the target program is located and which language frontend needs to be used to load its classes.

The **View** loads the elements of the target program only when they are queried, and memoizes them through configurable caching providers enabled by the new immutable IR design. The memoization is fine-grained, it works at the level of field, method, interface and modifier definitions. SOOTUP can create references to all of these objects via a corresponding language-specific **IdentifierFactory**. The references, i.e., the identifiers, are then used to access the queried elements of the target program.

Class types and signatures (for methods and fields) are considered global identifiers, across possibly concurrent instances of **Projects** and **Views**. They are created and pooled by the singleton instance of **IdentifierFactory** to reduce memory consumption. Additionally, it is cheaper to invoke `hashCode()` and `equals()` on the identifiers than on the IR objects that the identifiers reference.

2.4 Library by Default

SOOT had always been designed to be a standalone CLI (command-line interface) tool. This meant that it was expected to own the thread of control, which often hindered a tight integration of SOOT into integrated development environments (IDEs) or CI/CD pipelines, which are themselves frameworks and expect to own the thread of control as well. Also, a CLI aggregates all of the underlying functionality and makes it accessible via a single channel. This requires bundling everything together and contradicts our goals of providing lean modules.

To avoid this, we have conceptualized SOOTUP as a library by default. In SOOTUP, clients can depend on individual modules. For instance, to access the CFGs of a compiled program's methods, one needs to add a dependency to the *Java Bytecode Frontend* and *Core* modules. Further module dependencies can gradually be added later on when needed.

The library nature allows *the clients* to own the thread of control. This is preferable, especially, when using SOOTUP for other purposes than program analysis, or when using it as part of other analysis frameworks. SOOTUP also provides rather sophisticated functionality as a framework, with inversion of control, for instance when building call graphs or performing dataflow analyses.

Yet, SOOTUP is not quite stateless. As shown in Figure 3, the state is managed mainly by the `IdentifierFactory` and `View`. `View` instances keep references to all the memoized objects, they are not garbage collected unless the client releases the reference to the `View`. `IdentifierFactory`, on the other hand, maintains the global state of unique identifiers statically. It is the only singleton in SOOTUP, which might be shared across different views. In other words, if the client terminates then only the state in the `IdentifierFactory` will be retained.

2.5 Immutable IR by Design

SOOT was designed as a program optimization tool. Its main purpose was to enable the analysis and transformation of method bodies. As the research trend has shifted from program optimization to program analysis, we believe there is limited use in still maintaining mutable objects in a mutable IR.

Mutable objects are not easily shared between several entities. One needs to constantly account for unintended changes. They very much complicate parallelization at any level. To counter this problem, we have designed SOOTUP's JIMPLE IR to be immutable by default. This assures that there are no accidental modifications and that values can be safely shared and cached.

```

1  class Body {
2      ...
3
4      Body withStmts(List<Stmt> stmts) {
5          return new Body(stmts);
6      }
7  }
```

Listing 1.1. Modifying a Method Body via Withers

To ensure immutability we have slightly adjusted the API as well. Many classes do not have *setters* anymore, they have *withers* instead. Withers still allow modifications via new object copies with modified properties. Listing 1.1, for instance, shows how one can still modify the statements of a method body.

2.6 Changes to Jimple

Originally, JIMPLE was designed to be an IR for program optimization to fit SOOT's primary use case. Since the purpose of SOOTUP has been shifted to-

wards program analysis instead of optimization, we adjusted the JIMPLE IR towards this purpose. For efficiency reasons, a Java compiler compiles any `switch` statement to either a `tableswitch` or a `lookupswitch` bytecode instruction. Since the distinction is needed to transform the optimized JIMPLE back to bytecode, JIMPLE also made a distinction between `tableswitch` and `lookupswitch` statements. However, virtually all program analyses will treat both kinds of statements identically. Because of this, in SOOTUP both statements have been merged into a single `switch` statement, simplifying analysis implementations.

Another novelty in SOOTUP's JIMPLE is the added support for language extensibility. SOOTUP is designed to be an analysis framework that not just supports Java, but also other programming languages as well. To allow for this multi-language support, a basic JIMPLE IR has been implemented in a generic way that allows for easy extension with language-specific features. For the Java implementation, we extended this basic JIMPLE IR with import statements and annotations, two features that are highly specific to the Java language. Annotations are supported by extending JIMPLE's class type definition. Just like in Java source code, import statements improve the readability of Java-JIMPLE statements. Java-JIMPLE now allows referring to simple class names by defining their fully qualified names as imports. Likewise, basic JIMPLE can be extended to support features specific to other languages, e.g. JavaScript or Python.

3 Demonstration

In Section 2.3, we provided a glimpse of the new API. In this section, we demonstrate the new API with a set of most common use cases.

3.1 Setup

The code snippet in Listing 1.2 shows the starting point in SOOTUP to build an analysis project. The project builder requires two inputs: (1) the language of code to be analyzed and its version, as SOOTUP supports multiple languages; (2) the location of the analysis target. In this example, we are setting the analysis language as Java with version 8 and adding a Java classpath analysis input location that points to the analysis target. Note that one can add multiple analysis input locations to the project builder. The Java bytecode frontend accepts any of the Java archive formats (JAR or WAR), Android packages (APK), ZIPs or individual `.class` files. The Java source and the JIMPLE frontends accept `.java` and `.jimple` files respectively. To resolve a given class, the view will inspect all of the given analysis input locations.

```

1  JavaLanguage language = new JavaLanguage(8);
2  JavaProject project = JavaProject.builder(language)
3      .addInputLocation(new JavaClassPathAnalysisInputLocation("/path"))
4      .build();
5  JavaView view = project.createView();

```

Listing 1.2. The creation of a view in SOOTUP

3.2 Obtaining a Method Body

Assume the target code example in Listing 1.3. Following the API usage in Section 2.3, next we need to obtain a reference to the target class. To do so, as shown in Listing 1.4, we get the `IdentifierFactory` from the view at Line 6. We obtain the target class type at Line 7 and likewise the target method's signature at Line 8. A class is rather straightforward to identify, i.e. with a string corresponding to its *fully qualified name*, e.g. `"org.example.Main"` in this example.

Identifying methods requires a bit more information, as one needs to specify its containing class type, name, return type and parameter list to uniquely identify it. In this example, we use the target class type (`ct`) that we have created, set the name as `"run"` and return type as `"void"`. It is important to refer to any class type with its fully qualified name. For instance, while in Java it suffices to write `String[] args` to define the parameters as a string array, SOOTUP needs the definition as `java.lang.String[]`.

```
package org.example;
public class Main {

    void run(String[] args) {
        ...
    }
}
```

Listing 1.3. Target code example

```
6 IdentifierFactory factory = view.getIdentifierFactory();
7 ClassType ct = factory.getClassType("org.example.Main");
8 MethodSignature mSig = factory.getMethodSignature(
9   ct, "run", "void", Collections.singletonList("java.lang.String[]"));
```

Listing 1.4. Definition of a class type and a method signature using SOOTUP

The method signature that we created (`mSig`) can now be used to query the actual method object from the view. This is shown at Line 10 in Listing 1.5. As the new API follows the modern Java best practices, `view.getMethod()` returns an *optional*, at Line 11, we therefore test this optional for its presence and obtain the methods body. At Line 12, we output all the statements of the method.

```
10 view.getMethod(mSig)
11   .ifPresent(method ->method.getBody()
12     .getStmts().forEach(System.out::println));
```

Listing 1.5. Output all statements in a method body using SOOTUP

3.3 Call Graph Generation

A call graph models the calls between the methods of a target program, which makes it an essential data structure when performing interprocedural program analyses. SOOTUP's new call graph framework is based on a generic notion of a *CallGraphAlgorithm*, which can be extended by specific call graph algorithm implementations. The call graph algorithms only need to specify how they *resolve*

a call. Resolving can be based on the static class hierarchy (e.g. CHA [7], RTA [2]) or based on sophisticated pointer analyses [17].

```

13 CallGraphAlgorithm cha = new ClassHierarchyAnalysisAlgorithm(view);
14 CallGraph cg = cha.initialize(Collections.singletonList(mSig));
15 cg.containsMethod(anotherMethod)
16 cg.callsFrom(mSig)

```

Listing 1.6. Call graph generation using SootUP

Listing 1.6 shows an example of call graph generation using the new API. Since the view maintains all the classes and methods, it needs to be passed to the call graph algorithm, e.g. the `ClassHierarchyAnalysisAlgorithm` at Line 13. The call graph algorithm is initialized at Line 14, by specifying the entry method, which returns a `CallGraph` object. The call graph can be queried for method reachability, e.g. at Line 15, or can be iterated by retrieving the calls from the entry method, e.g. at Line 16.

3.4 Body Interceptors

Body interceptors in SootUP replace the concept of transformers in Soot. They essentially allow modifying method bodies, for instance, to add, remove or replace statements. As with the other objects, methods are immutable by default. Therefore, in SootUP any modifications to the method body must be performed during the body-building phase.

```

1 ClassLoadingOptions clo = new ClassLoadingOptions() {
2     @Override
3     public List < BodyInterceptor > getBodyInterceptors() {
4         return Collections.singletonList(new DeadAssignmentEliminator());
5     }
6 };
7 JavaView view = project.createView(analysisInputLocation -> clo);

```

Listing 1.7. Specifying Body Interceptors

Listing 1.7 shows an example of specifying a body interceptor. In this example the `DeadAssignmentEliminator` is specified. The body interceptors must be defined as part of the class loading options, as they are applied during class loading. The options are passed during the view creation.

4 Tool Support

Soot-based tools can be upgraded to use SootUP instead, however, depending on their implementation, the upgrading effort may vary. We next present the tools that SootUP currently supports and provides as submodules. We also suggest the roadmap for Soot-based tools for switching to SootUP.

4.1 Heros

HEROS [4] enables defining interprocedural dataflow analysis using the IFDS (interprocedural, finite, distributive subset) [24] and IDE (inter-procedural distributive environments) [25] conceptual frameworks. Both frameworks reduce dataflow analysis problems to graph reachability. While IDE well suits the analysis problems with large domains (such as tpestate or constant propagation analysis), IFDS is the primary choice for reachability analyses with a small domain (e.g. taint analysis).

```

1  JimpleBasedInterproceduralCFG icfg =
2      new JimpleBasedInterproceduralCFG(view, entryMethod);
3
4  IFDSTaintAnalysisProblem problem =
5      new IFDSTaintAnalysisProblem(icfg, entryMethod);
6
7  JimpleIFDSSolver<?, InterproceduralCFG<Stmt, SootMethod>> solver =
8      new JimpleIFDSSolver(problem);
9
10 solver.solve();

```

Listing 1.8. IFDS analysis using HEROS

SOOTUP provides the HEROS framework within its analysis submodule. Listing 1.8 shows an example on running an IFDS analysis using HEROS. SOOTUP implements HEROS' `InterproceduralCFG` interface with the JIMPLE-specific `JimpleBasedInterproceduralCFG`. To instantiate it, the client needs to pass the view and an entry method as shown at line 1. HEROS defines IFDS problems as an abstract class with `DefaultIFDSTabulationProblem`, this is extended by `DefaultJimpleIFDSTabulationProblem` in SOOTUP. However, the clients still need to define their custom IFDS analyses with problem-specific lattices, flow-functions and merge operators. An example of a basic IFDS-based taint analysis problem is available in SOOTUP, which is instantiated at line 4. SOOTUP extends HEROS' generic `IFDSSolver` with the `JimpleIFDSSolver` by concretizing it with `Stmt` (equivalent to `Unit` in SOOT) and `SootMethod`.

4.2 Qilin

Pointer information is an integral part of precise program analyses. SOOT's pointer analysis frameworks, SPARK [17] and its context-sensitive alternative PADDLE [18], have been popular in academia, as they provide a solid ground for researching novel algorithms. As we observe, however, the research trend is moving towards more sophisticated approaches with increased pointer analysis precision. For instance, context-sensitivity can be applied *selectively* rather than uniformly across the whole program [19].

QILIN [12] is a state-of-the-art flow-insensitive pointer analysis framework that was recently designed for supporting fine-grained selective context sensitivity while subsuming existing traditional method-level context sensitivity as

a special case. Since QILIN is fully written in Java and operates on the JIMPLE IR of SOOT, we were able to seamlessly incorporate QILIN into SOOTUP as a submodule with only minor engineering efforts. QILIN supports a rich set of pointer analyses such as Andersen’s context-insensitive analysis as implemented in SPARK [17], k -limiting callsite-sensitive analysis [27], k -limiting object-sensitive analysis [22, 28], and other recent advancements in pointer analysis. By providing QILIN as a SOOTUP submodule, we aim to foster comparative research using a broader set of pointer analysis algorithms.

```

1 PTAPattern ptaPattern = new PTAPattern("2o");
2 Collection entries = Collections.singleton(mainSig);
3 PTA pta = PTAFactory.createPTA(ptaPattern, view, entries);
4 pta.run();
5 CallGraph cg = pta.getCallGraph();

```

Listing 1.9. Call graph generation using a pointer analysis in QILIN

Listing 1.9 gives an example of 2-object sensitive pointer analysis using QILIN. In lines 1 and 2 the flavor of pointer analysis is specified and the entry method is set. In line 3 an instance of 2-object sensitive analysis is created which is subsequently executed in line 4. As the pointer analysis in QILIN supports on-the-fly call graph construction, the resulting call graph is retrieved in line 5. In addition, the pointer analysis API in QILIN provides `reachingObjects()`, for computing the points-to set of any variable and `mayAlias()`, for checking whether two variables are aliases. Note that QILIN is not part of SOOTUP’s current release.

4.3 Roadmap for Other Soot-based Tools

SOOTUP is not a drop-in replacement for SOOT. It is essentially a complete rewrite with a new architecture and API. We therefore primarily recommend SOOTUP to be used for new projects. However, existing tools that are based on SOOT can be upgraded to SOOTUP with some effort. The SOOTUP team has been working on upgrading some SOOT-based tools to SOOTUP. So far, we see that the roadmap, and thus the effort, for a specific tool to upgrade to SOOTUP will differ heavily based on how it is implemented. We have been seeing three recurring patterns: (1) generic tools that do not directly depend on SOOT, (2) tools that depend on SOOT but work with their own domain objects, (3) tools that depend on SOOT and work directly with SOOT objects.

Generic tools can swiftly be upgraded to SOOTUP. For instance, the API of the HEROS solver provides interfaces based on Java generics. Its interfaces can be extended with concrete tool-specific objects. The only requirement for SOOTUP to use the IFDS solver was to extend necessary interfaces by providing SOOTUP-specific objects.

Upgrading tools that use their own domain objects to SOOTUP is also simple. For instance, BOOMERANG [29] and SPARSEBOOMERANG [15], state-of-the-art demand-driven pointer analysis frameworks, implement their core functionality

within their own domain objects that correspond to classes, methods and statements. These tools require SOOTUP's objects to be converted to their domain objects via implementing an adapter.

Upgrading tools that work directly with SOOT objects is a more complex task. FLOWDROID [1], a popular Android information flow analysis tool, is highly intertwined with SOOT. It is hard to determine where exactly the boundaries of FLOWDROID are and how to separate it from SOOT. Therefore, at this point, we anticipate that FLOWDROID and tools of similar nature need a major rewrite to upgrade to SOOTUP. Nonetheless, we are considering upgrading even FLOWDROID to SOOTUP in the future.

5 Development

We next explain SOOTUP's development process, and how one can extend or contribute to SOOTUP.

5.1 SootUp's Development Process

We have incepted SOOTUP as a greenfield project. This choice not only granted us more freedom to restructure its architecture but also to employ a more modern software development process. Our new development process centers around continuous quality assurance. SOOT lacked proper test coverage, which complicated adding new features or any kind of nontrivial refactoring. To overcome this, we made testing an integral part of SOOTUP from the very beginning. SOOTUP is loaded with exhaustive unit and regression tests. We continuously observe its test coverage and enforce newly added code to maintain the same level of coverage. SOOTUP's tests currently account for 63.70% line coverage⁴ (9656 out of 15159 lines). To ensure that no new feature breaks or unintendedly changes SOOTUP's behavior, tests are executed for every new commit to SOOTUP's code repository through a continuous integration pipeline.

We seek to make SOOTUP more accessible to everyone. Our focus on an intuitive API design, as we explained in Section 2.3, is the first step in this direction. Further, we prioritize documentation and make it part of the development process. Our public-facing API elements are required to have Javadoc. Yet, we have learned, considering the questions in SOOT's issue tracker, that Javadoc alone is not enough. We thus maintain a documentation page⁵ to elaborate on some of the main concepts of SOOTUP's usage and provide more insight. To make the documentation beginner-friendly, we demonstrate the most common use cases with supporting code examples. From experience, we know that documentation tends to fall behind the most recent development state. To prevent this, we maintain the example code as part of SOOTUP's code repository. By doing so we ensure that the example code always compiles and functions with the most recent state.

⁴ <https://app.codecov.io/gh/soot-oss/SootUp>

⁵ <https://soot-oss.github.io/SootUp/>

SOOTUP is currently published at Maven Central. We have announced the first release (v1.0.0) in December 2022. Since then, we have been frequently releasing new features and bug fixes, the most recent version (v1.1.2) was published in June 2023. While, due to existing tool dependencies, SOOT and SOOTUP will coexist for a while, the bulk of our maintenance efforts will henceforth be directed toward SOOTUP rather than SOOT.

5.2 Extending and Contributing to SootUp

Concerning community engagement, SOOTUP will follow in the footsteps of SOOT. While SOOTUP’s development is currently still carried by Paderborn University, we are open for others to join the team. The main motivation behind our development efforts until the first release was to realize the design decisions laid out in Section 2. Since the first release, we have been focusing more on community feedback, such as bug reports and feature requests. Just like its predecessor, we expect SOOTUP to be shaped around the needs and contributions of the research community. We are eager to incorporate external contributions and very much welcome feature and pull requests. Repeat contributors may become core development team members with full commit rights.

To maintain an active community, we set up a discussion board on GitHub. This allows the community to participate in Q&As, suggest new ideas or simply discuss in an informal setting. SOOTUP is open-sourced with a GNU General Lesser Public License v2.1 (LGPL-2.1) [11]. It allows SOOTUP to be modified as long as the modifications are stated and licensed under the same license.

6 Future Work

SOOTUP is set to be the successor of the old SOOT framework. SOOT has been developed and improved for more than 20 years, so there are still multiple analysis utilities that need to be adapted to SOOTUP. Furthermore, we aim to keep up with advancements in the field of static program analysis and implement support for better callgraph construction approaches and more precise pointer-analysis techniques in SOOTUP as they are developed.

Being able to analyze Android applications was one of the main reasons for SOOT’s popularity. SOOTUP currently allows one to analyze Android applications with the help of dex2jar.⁶ This is an interim solution, as dex2jar is no longer actively maintained. In the meantime, we are working on a more robust solution based on Dexpler [3].

SOOTUP was designed with extensibility for other programming languages in mind. To allow for cross-boundary program analyses, we aim to implement new frontends for other languages. We especially aim at implementing a Python and a JavaScript frontend, due to the popularity of these languages. SOOTUP’s IR can be extended to cover at least other languages that, unlike C/C++, do

⁶ <https://github.com/pxb1988/dex2jar>

not allow direct pointer accesses. However, language-specific challenges are not out of the scope of this paper and need to be further investigated in the future.

Another goal for SOOTUP is to provide a means to enable the analysis of partial programs. To process an uncompiled Java source code project using SOOT or SOOTUP, the whole code base of the project, alongside all its dependencies, needs to be available either during compilation or during processing with the source code frontend. However, in some scenarios only part of the code base is available. In the future, we aim to provide support for processing such partial programs. By being able to generate Jimple from only partially available source code and substituting the missing information with either data that can be inferred from whatever is available of the code base or providing a means to additionally specify missing parts.

Performance comparison to SOOT or other tools was not possible because one would have to compare two identical analyses within these frameworks. Such analyses are still lacking at the moment. We, nevertheless, compared to SOOT on the unit test level. By design, SOOTUP shows significant performance improvements, particularly in class loading. The immutable IR was also designed to support much faster analyses than what is currently possible with SOOT's old JIMPLE IR. In the future, as SOOTUP-based analyses mature, we will conduct detailed performance evaluations.

In the future, we plan to also perform more evaluations regarding SOOTUP's usability. An API design that is as intuitive as possible for its users was one of the primary considerations when designing SOOTUP. To validate the API design, we plan to perform user studies with various types of user groups like researchers and software developers. Furthermore, we plan to benchmark SOOTUP's performance and compare it against other analysis frameworks and especially its predecessor.

7 Related Work

Apart from SOOT, there are various research-oriented static analysis frameworks. The most notable ones for Java are WALA [32], Doop [5] and OPAL [9]. WALA enables analyzing multiple programming languages such as Java, Javascript, and recently also Python [8]. It focuses on efficient static analysis by using specialized data structures. WALA's IR is close to JVM bytecode, but in contrast, it is based on SSA (static single assignment). Instead of operand stacks, it uses symbolic registers. SOOTUP is currently integrated with WALA's source code frontend, which enables SOOTUP to support source code in the same capacity as WALA does. Doop was originally developed as a pointer analysis framework. It enables defining static analyses declaratively and uses a Datalog solver. Doop's IR is also based on JIMPLE. It could probably be upgraded to SOOTUP with minor effort. OPAL provides highly configurable static analysis using abstract interpretation. PhASAR [26] is another notable static analysis framework that enables static analysis for C and C++ applications through the LLVM IR. LiSA [10] static analysis library enables novice users to implement static analyses that can target arbitrary languages based on the IMP programming language.

8 Conclusion

We have presented SOOTUP, a complete overhaul of the popular SOOT optimization and analysis framework for Java. SOOTUP shifts the purpose from optimization to static code analysis and fully modernizes the original SOOT implementation. SOOTUP implements all the lessons learned from the last 20+ years of development and usage of the original SOOT framework. It comprises many improvements like a new user-centric API, a fully parallelizable architecture and an new variant of the Jimple intermediate representation offering extensibility for multi-language support. With all these changes and improvements in place, SOOTUP aims to be a worthy successor of the good old SOOT framework and to enable the implementation of modern Java code analyses.

Acknowledgements. We gratefully acknowledge the contributions of Christian Brüggemann, Zun Wang, Andreas Dann, Marcus Nachtigall, Manuel Benz, Jan Martin Persch, Ben Hermann and Julian Dolby to SOOTUP's initial design and development. The development of SOOTUP was generously supported by the Research Software Sustainability funding line of the German Research Foundation (DFG) within the project FutureSoot, the Heinz Nixdorf Institute, and Amazon Web Services. It has also been partially funded by the German Federal Ministry of Education and Research (BMBF) through grant 01IS17046 Software Campus 2.0 (Paderborn University) as part of the project API_ASSIST. Responsibility for the content of this publication lies with the authors.

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* **49**(6), 259–269 (2014)
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. pp. 324–341 (1996)
3. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. pp. 27–38 (2012)
4. Bodden, E.: Inter-procedural data-flow analysis with ifds/ide and soot. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. pp. 3–8 (2012)
5. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. p. 243–262. OOPSLA '09, Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1640089.1640108>, <https://doi.org/10.1145/1640089.1640108>

6. Dann, A., Hermann, B., Bodden, E.: Sootdiff: Bytecode comparison across different java compilers. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. pp. 14–19 (2019)
7. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995* 9. pp. 77–101. Springer (1995)
8. Dolby, J., Shinnar, A., Allain, A., Reinen, J.: Ariadne: analysis for machine learning programs. In: *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. pp. 1–10 (2018)
9. Eichberg, M., Hermann, B.: A software product line for static analyses: The opal framework. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. p. 1–6. SOAP '14, Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2614628.2614630>, <https://doi.org/10.1145/2614628.2614630>
10. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. p. 1–6. SOAP 2021, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3460946.3464316>, <https://doi.org/10.1145/3460946.3464316>
11. Free Software Foundation, I.: Gnu lesser general public license v2.1 - gnu project - free software foundation. <https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html> (1999), (Accessed on 10/09/2023)
12. He, D., Lu, J., Xue, J.: Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In: Ali, K., Vitek, J. (eds.) *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 222, pp. 30:1–30:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). <https://doi.org/10.4230/LIPIcs.ECOOP.2022.30>, <https://drops.dagstuhl.de/opus/volltexte/2022/16258>
13. Hoe, A.V., Sethi, R., Ullman, J.D.: *Compilers—principles, techniques, and tools* (1986)
14. Karakaya, K., Bodden, E.: Sootfx: A static code feature extraction tool for java and android. In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. pp. 181–186. IEEE (2021)
15. Karakaya, K., Bodden, E.: Two sparsification strategies for accelerating demand-driven pointer analysis. In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. pp. 305–316. IEEE (2023)
16. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)* (Oct 2011), <https://www.bodden.de/pubs/lblh11soot.pdf>
17. Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.) *Compiler Construction*. pp. 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
18. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **18**(1), 1–53 (2008)

19. Li, Y., Tan, T., Möller, A., Smaragdakis, Y.: A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **42**(2), 1–40 (2020)
20. Li, Y., Tan, T., Zhang, Y., Xue, J.: Program tailoring: Slicing by sequential criteria. In: 30th European Conference on Object-Oriented Programming (ECOOP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
21. Luo, L., Bodden, E., Späth, J.: A qualitative analysis of android taint-analysis results. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 102–114. IEEE (2019)
22. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.* **14**(1), 1–41 (jan 2005). <https://doi.org/10.1145/1044834.1044835>, <https://doi.org/10.1145/1044834.1044835>
23. Piskachev, G., Krishnamurthy, R., Bodden, E.: Secucheck: Engineering configurable taint analysis for software developers. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 24–29. IEEE (2021)
24. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 49–61 (1995)
25. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* **167**(1–2), 131–170 (1996)
26. Schubert, P.D., Hermann, B., Bodden, E.: Phasar: An inter-procedural static analysis framework for c/c++. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 393–410. Springer (2019)
27. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) *Program Flow Analysis: Theory and Applications*, chap. 7, pp. 189–234. Prentice-Hall (1981)
28. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 17–30. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/1926385.1926390>, <https://doi.org/10.1145/1926385.1926390>
29. Späth, J., Nguyen Quang Do, L., Ali, K., Bodden, E.: Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In: 30th European Conference on Object-Oriented Programming (ECOOP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
30. Vallée-Rai, R., Gagnon, E.M., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing java bytecode using the soot framework: Is it feasible? In: International Conference on Compiler Construction (2000)
31. Vallée-Rai, R., Hendren, L.J.: Jimple: Simplifying java bytecode for analyses and transformations. Tech. rep., Technical report, McGill University (1998)
32. WALA: wala/wala: T.j. watson libraries for analysis, with frontends for java, android, and javascript, and may common static program analyses. <https://github.com/wala/WALA>, (Accessed on 10/04/2023)

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.





Formally verified asymptotic consensus in robust networks

Mohit Tekriwal, Avi Tachna-Fram, Jean-Baptiste Jeannin^(✉),
Manos Kapritsos, and Dimitra Panagou

University of Michigan, Ann Arbor, MI 48109, USA
{tmohit, avitf, jeannin, manosk, dpanagou}@umich.edu

Abstract. Distributed architectures are used to improve performance and reliability of various systems. Examples include drone swarms and load-balancing servers. An important capability of a distributed architecture is the ability to reach consensus among all its nodes. Several consensus algorithms have been proposed, and many of these algorithms come with intricate proofs of correctness, that are not mechanically checked. In the controls community, algorithms often achieve consensus *asymptotically*, e.g., for problems such as the design of human control systems, or the analysis of natural systems like bird flocking. This is in contrast to exact consensus algorithm such as Paxos, which have received much more recent attention in the formal methods community.

This paper presents the first formal proof of an asymptotic consensus algorithm, and addresses various challenges in its formalization. Using the Coq proof assistant, we verify the correctness of a widely used consensus algorithm in the distributed controls community, the *Weighted-Mean Subsequence Reduced (W-MSR) algorithm*. We formalize the necessary and sufficient conditions required to achieve resilient asymptotic consensus under the assumed attacker model. During the formalization, we clarify several imprecisions in the paper proof, including an imprecision on quantifiers in the main theorem.

Keywords: Resilient asymptotic consensus · W-MSR algorithm · Network robustness.

1 Introduction

To enhance reliability, robustness and performance, many modern systems use a distributed architecture, composed of multiple nodes communicating with each other. Examples range from coordinated control of multi-robot systems such as swarms of mobile and aerial robots, to load-balancing among servers answering many queries per second. A fully decentralized system, where decisions are made collectively by the nodes rather than by one master node, greatly improves reliability by ensuring there is no single point of failure in the system. A distributed architecture also provides greater performance (depending on the context, in terms of load capacity, reduced latency, smaller communication overhead, etc.)

than any single node could ever achieve. Distributed architectures are supported by distributed algorithms, which particularly focus on carefully handling situations where some nodes become faulty, stop responding, or become malicious.

One central aspect of distributed algorithms is the ability to achieve *consensus*. Consensus is said to be achieved in a network if all normal (correct) nodes agree on a certain value, where a node is *normal* if it is not faulty [34]. The value agreed upon by all nodes can be a reference point for the next position of a swarm, or the sequence of commands executed by a set of replicas in State Machine Replication [44]. Consensus has been studied extensively in different communities. In the distributed computer systems communities, some prominent algorithms achieving consensus are Paxos [29], MultiPaxos [47], Raft [36], and Practical Byzantine Fault Tolerance (PBFT) [6]. However, these algorithms deal with the problem of exact consensus. There are many scenarios where exact consensus is not achievable, ranging from the design of human controlled systems to analysis of natural systems like bird flocking. These problems have to be solved under harsh environmental restrictions such as restricted communication abilities and presence of communication uncertainty. Therefore, these problems warrant the study of *asymptotic consensus* problems, which unlike exact consensus, do not require strong assumptions on the underlying network [16].

This paper presents the first formal proof of an asymptotic consensus algorithm, by formalizing the Weighted-Mean Subsequence Reduced (W-MSR) algorithm [30, 50]. The problem of asymptotic consensus is of much importance to the distributed robotics and controls community, who have studied algorithms like the Mean Subsequence Reduced (MSR) algorithm [27] and its recent extension W-MSR. These algorithms are designed to achieve asymptotic consensus in partially connected groups of nodes, but have not been formally verified. Formal verification of consensus algorithms is important as has been emphasized by the distributed computer systems community, who have long invested in producing mechanically checked proofs of its consensus protocols. The controls community, however, lags behind in this direction. In recent years, the distributed systems community has embraced formal methods to provide *mechanically-checked* proofs of its consensus protocols and their implementations, using a wide range of techniques from interactive and automated theorem proving [48, 25, 8, 5, 18, 9, 31] to automatic generation of inductive invariants [33, 21, 49, 20]. In the distributed robotics and controls community however, researchers usually prove their consensus protocols with paper proofs, using mathematical analysis based on Lyapunov theory and its extensions, without computer-checked formalizations. As we show in this paper, our formalization of asymptotic consensus for the W-MSR algorithm [30] reveals imprecisions in the placement of quantifiers in the main theorem and several missing pieces in the proof, thereby highlighting the importance of machine-checked proofs. Thus a significant contribution of our work is providing the first mechanically checked formalism of the asymptotic consensus and its application to the W-MSR algorithm, widely used in the controls community. We have chosen to formalize this algorithm since it is a widely-used algorithm for resilient consensus [42, 41, 46]. From the perspective

of practical applications, enabling resilient consensus in the presence of misbehaving or faulty nodes is desirable for many applications in autonomous systems and robotics, e.g., for coordinated control of multi-robot systems.

The MSR and W-MSR algorithms are very different from exact consensus algorithms such as MultiPaxos, Raft or PBFT. As such our formal verification of the correctness of W-MSR uses different techniques than previous proofs of exact consensus algorithms. The first major difference is that MSR and W-MSR guarantee *asymptotic* consensus rather than finite-time consensus. A second major difference is that MSR and W-MSR provide consensus in networks that are *not fully connected*: two normal nodes might not be able to communicate with each other directly, but might have to rely on another (possibly faulty) node to forward their messages to each other. This last property is crucial to model multi-robot systems where complete communication between any two robots may not be feasible at all times. Because of those differences, providing a mechanically-checked proof of W-MSR requires the development and use of different techniques than the ones typically used to mechanically check Multipaxos, Raft or PBFT. In particular, our formalization crucially relies on formalization of limits and real analysis, because many of the techniques used in model-checking or for generating invariants are not well-suited to prove asymptotic properties.

Contributions: The original contribution of this work is the formalization in the Coq theorem prover of the convergence results of the W-MSR algorithm [30]. Specifically, we provide a machine-checked concrete counterexample for the proof of necessity, a clean proof of Lemma 1 and the Coq formalization of the main theorem (Theorem 1). We also fill in several missing details and clarify imprecisions in the proof of sufficiency, which can be viewed as an addition to the existing proof [30]. Additionally, this is, to our knowledge, the first mechanical formalization of a consensus algorithm where the consensus is obtained asymptotically, opening the door to more such proofs.

This paper is organized as follows. In Section 2, we discuss the problem setup and define terminologies related to graph topology and the W-MSR algorithm [30]. In Section 3, we discuss the formalization of the necessary and sufficient conditions in Coq, for achieving resilient asymptotic consensus. We also discuss some specific challenges we encountered during the formalization. After reviewing some related work in Section 4, we conclude in Section 5 by discussing key takeaways from our work and generic challenges we encountered during the formalization. We also lay down a few directions that could be addressed in future work.

2 Preliminaries

In this paper we consider the problem of formalizing consensus in a network, and adopt the problem formulation from [30]. While the original paper discusses consensus in a distributed control graph for both malicious and byzantine threat models for both time-varying and time-invariant graph structures, we limit our formalization to the case of a *time-invariant graph* for a *malicious threat model* and for a particular threat scope: *F-total*, where the total number of malicious

nodes in the control graph is bounded. We will next discuss briefly what each of these highlighted terms means in the context of the following problem.

2.1 Problem formulation

Consider a network that is modeled by a *digraph* (directed graph), $\mathcal{D} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \dots, n\}$ is the *node set* and $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is the *directed edge set*. The node set is partitioned into a set of *normal nodes* \mathcal{N} , and a set of *adversary nodes* \mathcal{A} , which are unknown a priori to the normal nodes. Each directed edge $(j, i) \in \mathcal{E}$ models *information flow* and indicates that node i can be influenced by (or receive information from) node j at time-step t . The set of *in-neighbors* of node i is defined as $\mathcal{V}_i = \{j \in \mathcal{V} \mid (j, i) \in \mathcal{E}\}$. Intuitively, the set of in-neighbors contains all neighboring nodes of i , such that the direction of information flow is from those nodes to i . The cardinality of the set of in-neighbors is called the *in-degree*, $d_i = |\mathcal{V}_i|$. Since each node has access to its own value at time-step t , we also consider a set of *inclusive neighbors* of node i , denoted by $\mathcal{J}_i = \mathcal{V}_i \cup \{i\}$.

2.2 Threat Model

As discussed earlier, we formalize a threat model (*F-total malicious model* [30]) in which every adversary node in the graph is *malicious*, and there exists an upper bound F on the number of malicious agents in the graph, i.e., the set of adversary nodes are *F-totally bounded*. In the context of the problem in Section 2.1, some relevant formal definitions pertaining to the threat model are stated as:

Definition 1 (Malicious node [30]). A node $i \in \mathcal{A}$ is called **Malicious** if it sends the same value $x_i(t)$ to all its neighbors at each time step t , but applies a different update function $f'_i(\cdot)$ at some time step.

Definition 2 (F-total set [30]). A set $\mathcal{S} \subset \mathcal{V}$ is **F-total** if it contains at most F nodes in the network, i.e., $|\mathcal{S}| \leq F$, $F \in \mathbb{Z}_{\geq 0}$.

Definition 3 (F-totally bounded [30]). A set of adversary nodes is **F-totally bounded** if it is an *F-total set*.

Note that while Definitions 2 and 3 may appear similar, they define different terminologies. Definition 2 defines an *F-total set* with at most F nodes in a network. Definition 3 specializes this to a set of adversary nodes saying that there are at most F adversarial nodes in a network.

2.3 Robust network topologies

The ability of a set of normal nodes in a control graph to achieve consensus depends on its ability to make local decisions effectively. Le Blanc et al. [30] defined a topological property called *network robustness* for reasoning about the effectiveness of purely local algorithms to succeed, which we formalize in Coq. In particular, they define a property called (r, s) -robustness, which is stated as:

Definition 4 ((r, s) -robustness [30]). : A digraph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ on n nodes ($n \geq 2$) is (r, s) -robust, for nonnegative integers $r \in \mathbb{Z}_{\geq 0}$, $1 \leq s \leq n$, if for every pair of nonempty, disjoint subsets \mathcal{S}_1 and \mathcal{S}_2 of \mathcal{V} at least one of the following holds (i) $|\mathcal{X}_{\mathcal{S}_1}^r| = |\mathcal{S}_1|$; (ii) $|\mathcal{X}_{\mathcal{S}_2}^r| = |\mathcal{S}_2|$; (iii) $|\mathcal{X}_{\mathcal{S}_1}^r| + |\mathcal{X}_{\mathcal{S}_2}^r| \geq s$, where $\mathcal{X}_{\mathcal{S}_k}^r = \{i \in \mathcal{S}_k : |\mathcal{V}_i \setminus \mathcal{S}_k| \geq r\}$ for $k \in \{1, 2\}$.

The condition (iii) states that there are a total of at least s nodes from the union of sets \mathcal{S}_1 and \mathcal{S}_2 , such that each of those nodes have at least r nodes outside of their respective sets in the union $\mathcal{S}_1 \cup \mathcal{S}_2$. The idea is that “enough” nodes in every pair of nonempty, disjoint sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathcal{V}$ have at least r neighbors outside of their respective sets. This ensures that the network is well connected, and that loss of information from a node due to malicious attack does not affect the whole network. Figure 1 illustrates an example of a network with $(2, 2)$ robustness.

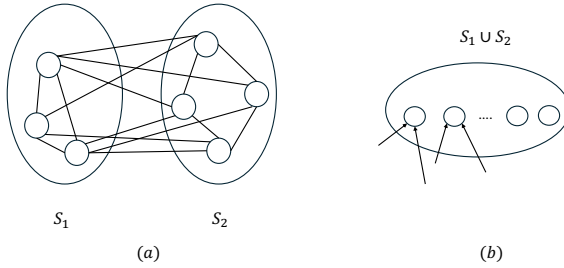


Fig. 1. Illustration for $(2, 2)$ robustness. In the illustration (a), every node of the set \mathcal{S}_2 has 2 neighboring nodes outside \mathcal{S}_2 . Similarly every node in the set \mathcal{S}_1 has at least 2 neighboring nodes outside \mathcal{S}_1 . In the illustration (b), there are 2 nodes in the union $\mathcal{S}_1 \cup \mathcal{S}_2$ that have 2 neighbors outside the set. Note that the sets \mathcal{S}_1 and \mathcal{S}_2 are disjoint.

2.4 Update model for the normal nodes

In this paper, we formalize a consensus algorithm, called the W-MSR algorithm [30]. This algorithm provides an update model for the normal nodes in the network. A schematic of the algorithm is illustrated in Figure 2. We denote the value emitted by node i at time t as $x_i(t)$, and the value of the directed weighted edge from node j , to node i at time t as $w_{ij}(t)$. The value $x_i(t)$ could represent a measurement like position, velocity, or it could be an optimization variable. The quantity $x_j^i(t)$ is the information that the j^{th} node in the neighboring set of node i sends to the node i . Each node also has a varying set of neighbors which it ignores that we denote as $\mathcal{R}_i(t)$. The set $\mathcal{R}_i(t)$ changes because the nodes are removed depending on their value with respect to the value of node i at time t . In this algorithm, the updated value of a normal node i at time $t + 1$ is the convex sum of the values of its neighboring set including itself. Hence, $x_i(t + 1) = \sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} w_{ij}(t) x_j^i(t)$, where we assume the existence of a constant $\alpha \in \mathbb{R}$, such that $0 < \alpha < 1$, and the weights $w_{ij}(t)$ satisfy the conditions:

1. $w_{ij}(t) = 0$ whenever $j \notin \mathcal{J}_i$;
2. $w_{ij}(t) \geq \alpha, \forall j \in \mathcal{J}_i$; and
3. $\sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} w_{ij}(t) = 1$

for all $i \in \mathcal{N}$, and $t \in \mathbb{Z}_{\geq 0}$. It is important to note that the third condition depends on the set of removed nodes, which may change over time. In order to satisfy this condition the values of the weights may need to change over time.

The choice of neighboring sets in the W-MSR algorithm is defined as follows:

1. At each time-step t , each normal node i obtains the values of its neighbors, and forms a sorted list
2. If there are fewer than F nodes with values strictly greater than the value of i , then the normal node removes all those nodes. Otherwise, it removes precisely the largest F values in the sorted list. Likewise, if there are less than F nodes with values strictly less than the normal node i , the normal node removes all such nodes. Otherwise, it removes precisely the smallest F nodes in the sorted list.

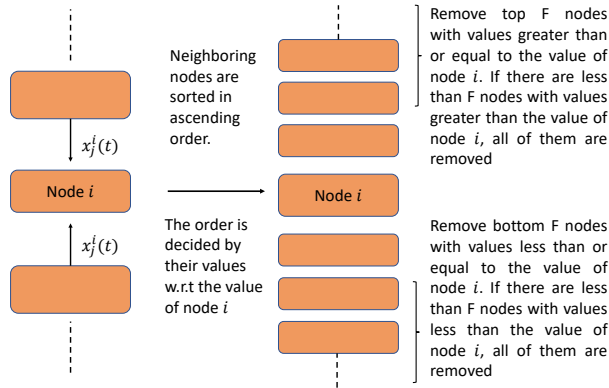


Fig. 2. Schematic of the W-MSR update. At time t , the node i obtains values from its neighbors and forms a sorted list. The algorithm then removes the largest and the smallest F nodes in the sorted list, or if there are less than F nodes with values strictly greater than or less than the value of i , the algorithm removes all those nodes.

An important point to note here is that the above update model holds only for the normal nodes, i.e., $i \in \mathcal{N}$. The update function for adversary nodes, i.e. $i \in \mathcal{A}$, and their influence on the normal nodes depend on the threat model. We will next discuss the formalization of the W-MSR algorithm in Coq.

3 A formal proof of consensus for the W-MSR algorithm

Theorem 1. [30] Consider a time-invariant network modeled by a digraph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ where each normal node updates its value according to the W-MSR algorithm with parameter F . Under the F -total malicious model, resilient asymptotic consensus is achieved if and only if the network topology is $(F + 1, F + 1)$ -robust.

The proof of this theorem requires us to prove both a sufficiency and a necessity condition. The original paper proof relies on a safety condition, which provides an invariant condition that must hold at all times in the state update. We will next discuss the proof of the safety condition (Section 3.1), then sufficiency (Section 3.2) and necessity (Section 3.3) conditions individually.

3.1 Proof of the safety condition in W-MSR

Lemma 1 (Safety condition). [30] *Suppose each node updates its value according to the W-MSR algorithm with parameter F under the F -total malicious model. Then for each node $i \in \mathcal{N}$, $x_i(t+1) \in [m(t), M(t)]$, regardless of the network topology.*

Here, $m(t) = \min_{i \in \mathcal{N}} \{x_i(t)\}$ and $M(t) = \max_{i \in \mathcal{N}} \{x_i(t)\}$. Note that the original paper [30] does not provide a proof of this lemma, and our proof, which we formalize in this paper, is an original contribution. We provide a detailed proof of the lemma by explicitly enumerating the cases from the definition of the W-MSR algorithm. On the other hand, the original paper [30] merely states an outline, making a careful check of the proof difficult.

Proof. We prove Lemma 1 by showing inductively, that at each time t , and for every normal node i , there exists a node $j_1 \in \mathcal{J}_i \cap \mathcal{N}$ such that $\forall k \in \mathcal{J}_i \setminus \mathcal{R}_i(t)$, $x_{j_1}(t) \leq x_k(t)$, thus:

$$x_i(t+1) = \sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} w_{ij}(t) x_j^i(t) \geq \sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} w_{ij}(t) x_{j_1}^i(t) = x_{j_1}^i(t) \geq m(t) \quad (1)$$

Symmetrically there exists a $j_2 \in \mathcal{J}_i \cap \mathcal{N}$ such that $\forall k \in \mathcal{J}_i \setminus \mathcal{R}_i(t)$, $x_{j_2}(t) \geq x_k(t)$. Thus, the symmetric inequality $x_i(t+1) \leq M(t)$, holds for the same reason. Since the proof of the existence of j_1 and j_2 are nearly identical, we only show the proof of the former in Appendix A of the extended version [45].

Formalization in Coq: We formalize Lemma 1 in Coq as:

```

Lemma lem_1:  $\forall (i:D) (t:\text{nat}) (\text{mal}:\text{nat} \rightarrow D \rightarrow R) (\text{init}:D \rightarrow R)$ 
 $(A:D \rightarrow \text{bool}) (w:\text{nat} \rightarrow D * D \rightarrow R),$ 
F_total_malicious mal init A w  $\rightarrow$ 
wts_well_behaved A mal init w  $\rightarrow$ 
 $i \in \text{Normal } A \rightarrow ((x \text{ mal init } A w (t+1) i \leq M \text{ mal init } A w t)$ 
 $\wedge (m \text{ mal init } A w t \leq x \text{ mal init } A w (t+1) i)).$ 

```

The definition of **F_total_malicious** states that the model is F -total malicious if the set of adversary nodes are F -totally bounded (i.e., there are at most F adversary nodes in the network) and all the adversary nodes are malicious. Here $A : D \rightarrow \text{bool}$ is a tagging function. If $A \ i == \text{true}$, then i is classified as an *Adversary* node else it is classified as a *Normal* node. $\text{mal} : \text{nat} \rightarrow D \rightarrow R$ is an arbitrary update function for a malicious node. Since we do not know beforehand, how this function would look like, we assume it as a parameter. The function $\text{init} : D \rightarrow R$ is an initial value associated with a node. We define a **malicious** node in Coq as that node in the graph for which the normal update model does not hold, i.e., there exists a time t such that $x_i(t+1) \neq \sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} w_{ij}(t) x_j^i(t)$.

```

(** Condition for a node to have malicious behavior at a given time **)
Definition malicious_at_i_t (mal:nat → D → R) (init:D → R) (A:D → bool)
(w:nat → D * D → R) (i:D) (t:nat): bool :=
(x mal init A w (t+1) i) !=  $\sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} ((x \text{ mal init A w t j}) * (w \text{ t (i,j)}))$ 

```

```

(** Define maliciousness **)
Definition malicious (mal:nat → D → R) (init:D → R) (A:D → bool)
(w:nat → D * D → R) (i:D) :=  $\exists t:\text{nat}, \text{malicious\_at\_i\_t mal init A w i t}$ .

```

The second hypothesis `wts.well behaved` states that we respect those three conditions on weights that we discussed in Section 2.4. The assignment of weights depend on whether a node $j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)$ or not. Here, \mathcal{J}_i denotes the inclusive set of neighbors of the node i . $\mathcal{R}_i(t)$ denotes the removed set of nodes according to the W-MSR algorithm, and we define $\mathcal{R}_i(t)$ in Coq as follows

```

Definition remove_extremes (i:D) (l:seq D) (x:D → R) : (seq D) :=
  filter (fun (j:D) =>
    (((Rge_dec (x j) (x i)) || (F ≤ (index j l))) && (Rle_dec (x j) (x i)
    || (index j l ≤ ((size l) - F - 1)))) l.

```

Note that we use the filter function from the `MathComp` sequence library. This is crucial as it gives us lemmas that allow us to assert that any node in $\mathcal{J}_i \setminus \mathcal{R}_i(t)$ satisfies the conditions of the filter. Additionally, the filter function requires that its first argument has a `pred` type, $D \rightarrow \text{bool}$ in our case. Therefore, we need our inequality operations to be decidable. Hence, we used the decidable versions of the inequality operations, such as `Rle_dec`, provided by Coq's `reals` library instead of its built-in \leq operation. We then define the set $\mathcal{J}_i \setminus \mathcal{R}_i(t)$ in Coq as

```

Definition incl_neigh_minus_extremes
(i:D) (x:D → R) : (seq D) := remove_extremes i (inclusive_neighbor_list i x) x.

```

Since $\mathcal{J}_i \setminus \mathcal{R}_i(t)$ is defined based on the value of node i , $x_i(t)$, which indeed depends on `A`, `mal`, `init`. Hence, `wts.well behaved` depends on `A`, `mal`, `init`.

The trickiest parts of the proof of Lemma 1 rely on the fact that we desire $\mathcal{J}_i \setminus \mathcal{R}_i(t)$ when treated as a list to be sorted. In order to fulfill this condition we use the formalization for sorting found in the `MathComp` library. To do this we first define a relation on D as:

```

Definition sorted_Dseq_rel (x: D → R) (i j : D) :=
  if Rle_dec (x i) (x j) then
    if (x i = x j) then (index i (enum D) ≤ index j (enum D)) else true
  else false.

```

This definition ensures that if $x_i(t) < x_j(t)$, then i is ordered as less than j with respect to this relationship. In the case of nodes with equivalent values we use an arbitrary mechanism to break ties. Doing so ensures that this relation is total, and satisfies transitivity, anti-symmetry, and reflexivity. This relation lets us use the sorting lemmas in `MathComp`'s `path` library [13], and it ensures the weaker condition that we occasionally use in the proof:

```

Definition sorted_Dseq (x:D → R) (l:seq D) :=
 $\forall (a b:D), a \in l \rightarrow b \in l \rightarrow (\text{index } a \text{ l} < \text{index } b \text{ l}) \rightarrow (x \text{ a} \leq x \text{ b})$ .

```

The biggest difficulty with formalizing this proof arises when dealing with the case that $|R_i^<(t)| < F$, where $R_i^<(t) := \{j \in \mathcal{J}_i : x_j(t) < x_i(t) \text{ and } \text{idx}_{\mathcal{J}_i}(x_j(t)) < F\}$, and define $\text{idx}_l(x_k(t))$, to be the index of the value $x_k(t)$ in a given list l of values, or the size of l if $x_k(t)$ is not present.. In particular, showing that $\text{idx}_{\mathcal{J}_i \setminus \mathcal{R}_i(t)}(j) = 0 \implies n_j(\mathcal{J}_i) = |R_i^<(t)|$. This requires proving an extra lemma on the \mathcal{J}_i list:

```
Lemma partition_incl:  $\forall (i:D) (t:\text{nat}) (\text{mal}:\text{nat} \rightarrow D \rightarrow R)$ 
   $(\text{init}:D \rightarrow R) (A:D \rightarrow \text{bool}) (w:\text{nat} \rightarrow D * D \rightarrow R),$ 
  inclusive_neighbor_list i (x mal init A w t) =
  (sort ((sorted_Dseq_rel (x mal init A w t))) )
    (enum (R_i_less_than mal init A w i t))) + +
  (incl_neigh_minus_extremes i (x mal init A w t)) + +
  (sort ((sorted_Dseq_rel (x mal init A w t))) )
    (enum (R_i_greater_than mal init A w i t))).
```

With this lemma, we can reason that the zero-th index of $\mathcal{J}_i \setminus \mathcal{R}_i(t)$, is the $|R_i^<(t)|$ -th index of \mathcal{J}_i . Using this lemma, we can prove the existence of j_1 in the proof of `lem_1`. Symmetrically, we can show the existence of j_2 such that $\forall k \in \mathcal{J}_i \setminus \mathcal{R}_i(t), x_{j_2}(t) \geq x_k(t)$. Tying it all together, we complete the proof of the lemma `lem_1` in Coq.

3.2 Proof of Sufficiency

Lemma 2. [30] *Consider a time-invariant network modeled by a digraph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ where each normal node updates its value according to the W-MSR algorithm with parameter F . Under the F -total malicious model, if a network is $(F+1, F+1)$ robust, resilient asymptotic consensus is achieved.*

This is an important lemma because we would like to design a network such that the normal nodes in the network reach an asymptotic consensus in the presence of malicious nodes in the network. Next we will discuss an informal proof of the Lemma 2 followed by its formalization in the Coq proof assistant.

Proof. The proof of Lemma 2 is done by contradiction. We start by assuming that the limits A_M and A_m of the functions $M(t)$ and $m(t)$ respectively are different, i.e., $A_M \neq A_m$. The limits A_M and A_m of the functions $M(t)$ and $m(t)$, respectively, exist because $M(t)$ and $m(t)$ are both continuous and monotonously decreasing functions of t . Therefore, by definition of limits for $M(t)$ and $m(t)$, we know that $\forall t, A_M \leq M(t) \wedge m(t) \leq A_m$, as illustrated in Figure 3. We will show that by carefully constructing the sets S_1 and S_2 in the definition of (r, s) -robustness, and unrolling the definition of (r, s) -robustness at every time-step inductively, we eventually arrive at the desired contradiction: $\exists t, M(t) < A_M \vee A_m < m(t)$. We discuss the details of the proof in Appendix B of the extended version [45].

Formalization in Coq: We introduce the following axiom in Coq to support reasoning by contradiction.

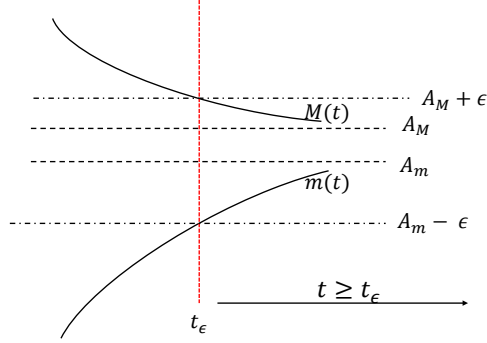


Fig. 3. Illustration of the tube of convergence bounded above by $A_M + \epsilon$ and bounded below by $A_m - \epsilon$. We observe the behavior of functions $M(t)$ and $m(t)$ inside this tube of convergence $\forall t \geq t_\epsilon$. We prove that $M(t)$ and $m(t)$ are monotonous $\forall t \geq t_\epsilon$, and they approach the limits A_M and A_m , respectively. We start by assuming that $A_M \neq A_m$, but later prove that $A_M = A_m$ by contradiction, thereby proving asymptotic consensus.

Axiom `proposition_degeneracy` : $\forall A : \text{Prop}, A = \text{True} \vee A = \text{False}$.

This is a propositional completeness lemma that allows us to reason classically and is consistent with the formalization of classical facts in Coq's standard library. We need this lemma because we prove the sufficiency condition using contradiction. We are choosing to use classical reasoning because the original paper [30] does not provide a constructive proof. The reasoning used in the paper is classical. This requires us to state the following lemma in Coq

Lemma `P_not_not_P`: $\forall (P : \text{Prop}), P \leftrightarrow \neg(\neg P)$.

The proof of `P_not_not_P` uses the axiom `proposition_degeneracy`.

We state the sufficiency condition (Lemma 2) for the network to achieve resilient asymptotic consensus as the following in Coq.

Lemma `strong_sufficiency`:

```

 $\forall (A : D \rightarrow \text{bool}) (\text{mal} : \text{nat} \rightarrow D \rightarrow \text{R}) (\text{init} : D \rightarrow \text{R}) (\text{w} : \text{nat} \rightarrow D * D \rightarrow \text{R}),$ 
nonempty_nontrivial_graph  $\rightarrow$ 
 $(0 < F + 1 \leq |D|) \% N \rightarrow$ 
wts_well_behaved  $A \text{ mal init w} \rightarrow$ 
r_s_robustness  $(F + 1) (F + 1) \rightarrow$ 
Resilient_asymptotic_consensus  $A \text{ mal init w}$ .

```

The sufficiency condition requires that the graph is non-trivial, i.e., there are at least two nodes in the graph, and the number of faulty nodes F in the graph is bounded by the total number of nodes D . We define `r_s_robustness` in Coq as

Definition `r_s_robustness` ($r \ s : \text{nat}$):= `nonempty_nontrivial_graph` $\wedge ((1 \leq s \leq |D|) \rightarrow$

$\forall (S1\ S2: \{\text{set } D\}),$
 $(S1 \subset \text{Vertex} \wedge (|S1| > 0)) \rightarrow$
 $(S2 \subset \text{Vertex} \wedge (|S2| > 0)) \rightarrow$
 $[\text{disjoint } S1 \ \& \ S2] \rightarrow$
 $((|Xi_S_r\ S1\ r| = |S1|) \vee (|Xi_S_r\ S2\ r| = |S2|) \vee$
 $(|Xi_S_r\ S1\ r| + |Xi_S_r\ S2\ r| \geq s)) \vee$.

where $Xi_S_r\ S1\ r$ is the set of all nodes in the set $S1$ such that all of its nodes have at least r neighboring nodes outside $S1$. In Coq, we define Xi_S_r as

Definition $Xi_S_r\ (S: \{\text{set } D\})\ (r: \text{nat}):=$
 $[\text{set } i: D \mid i \in S \ \& \ (|(\text{in_neighbor } i) - S| \geq r)]$.

We define `Resilient_asymptotic_consensus` in Coq as

Definition `Resilient_asymptotic_consensus`
 $(A: D \rightarrow \text{bool})\ (mal: \text{nat} \rightarrow D \rightarrow R)\ (init: D \rightarrow R) :=$
 $(F_total_malicious\ mal\ init\ A\ w) \rightarrow (\exists\ L: Rbar, \forall (i: D),$
 $i \in (\text{Normal } A) \rightarrow \text{is_lim_seq } (\text{fun } t: \text{nat} \Rightarrow x\ mal\ init\ A\ w\ t\ i)\ L) \wedge$
 $(\forall\ t: \text{nat}, (m\ mal\ init\ A\ w\ 0 \leq m\ mal\ init\ A\ w\ t) \wedge$
 $(M\ mal\ init\ A\ w\ t \leq M\ mal\ init\ A\ w\ 0))$.

Here, `is_lim_seq` is a predicate in Coquelicot that defines limits of sequences. $Rbar$ is the extended set of reals, which includes $+\infty$ and $-\infty$. To prove that the network achieves resilient asymptotic consensus under the $(F+1, F+1)$ -robustness condition, we need to prove the following two conditions in the definition of `Resilient_asymptotic_consensus`: (i) $\forall t, m(0) \leq m(t) \wedge M(t) \leq M(0)$, and (ii) $\exists L, \forall i, i \in \mathcal{N} \rightarrow \lim_{t \rightarrow \infty} x_i(t) = L$. We state the first subproof as the lemma statement `interval_bound` in Coq. The proof of lemma `interval_bound` is a consequence of Lemma 1. We prove this lemma by an induction on time t and then apply Lemma 1 to complete the proof.

We prove the second subproof by contradiction in Coq. To start the proof of contradiction, we need to assume that the limits A_M and A_m of the maximum and minimum functions $M(t)$ and $m(t)$ are different. We then instantiate the sets S_1 and S_2 in the definition of (r, s) -robustness with $\mathcal{X}_M(t_\epsilon, \epsilon_o)$ and $\mathcal{X}_m(t_\epsilon, \epsilon_o)$ respectively, where $\mathcal{X}_M(t, \epsilon_l) = \{i \in \mathcal{V} : x_i(t) > A_M - \epsilon_l\}$ and $\mathcal{X}_m(t, \epsilon_l) = \{i \in \mathcal{V} : x_i(t) < A_m + \epsilon_l\}$. In Coq, we define the sets \mathcal{X}_M for any epsilon and t as follows

Definition $X_m_t_e_i\ (e_i: R)\ (A_m: R)\ (t: \text{nat})\ (mal: \text{nat} \rightarrow D \rightarrow R)\ (init: D \rightarrow R)$
 $(A: D \rightarrow \text{bool})\ (w: \text{nat} \rightarrow D \rightarrow R) :=$
 $[\text{set } i: D \mid \text{Rlt_dec } (x\ mal\ init\ A\ w\ t\ i)\ (A_m + e_i)]$.

where `Rlt_dec` is Coq's standard decidability lemma for less than operation.

We need to prove that the sets \mathcal{X}_M and \mathcal{X}_m are disjoint at all times till we reach a point when either \mathcal{X}_M or \mathcal{X}_m are empty. This requires us to prove the following lemma in Coq

Lemma `X_M_X_m_disjoint_at_j`
 $(mal: \text{nat} \rightarrow D \rightarrow R)\ (init: D \rightarrow R)\ (A: D \rightarrow \text{bool})\ (w: \text{nat} \rightarrow D \rightarrow R):$
 $\forall (t_eps\ l: \text{nat})\ (a\ A_M\ A_m: R)\ (eps_0\ eps: \text{posreal}),$

$$\begin{aligned}
& (A_m - (\text{eps_j } 1 \text{ eps_0 eps a}) > A_m + (\text{eps_j } 1 \text{ eps_0 eps a}) \rightarrow \\
& [\text{disjoint } (X_m_t_e_i (\text{eps_j } 1 \text{ eps_0 eps a}) A_m (t_eps+1) \text{ mal init A w}) \ \& \\
& (X_m_t_e_i (\text{eps_j } 1 \text{ eps_0 eps a}) A_m (t_eps+1) \text{ mal init A w})].
\end{aligned}$$

Since $\mathcal{X}_m(t_\epsilon + l, \epsilon_l)$ is a set of all nodes with values at least, $A_m - \epsilon_l$ and $\mathcal{X}_m(t_\epsilon + l, \epsilon_l)$ is a set of all nodes with values at most $A_m + \epsilon_l$, these two sets are disjoint if $A_m - \epsilon_l > A_m + \epsilon_l$. For $l = 0$, we have defined ϵ_o such that $A_m - \epsilon_o > A_m + \epsilon_o$. To prove that $A_m - \epsilon_l > A_m + \epsilon_l, \forall l, 0 < l$, we need to show that $A_m - \epsilon_l > A_m - \epsilon_o$ and $A_m + \epsilon_o > A_m + \epsilon_l$. This would indeed require us to show that $\epsilon_l < \epsilon_o, \forall l, 0 < l$. This holds since we had defined ϵ_l recursively as $\epsilon_l := \alpha \epsilon_{l-1} + (1 - \alpha)\epsilon$.

A crucial aspect of the sufficiency proof is proving that the $(F + 1, F + 1)$ -robustness implies that there exists a node in the union of the set $\mathcal{X}_M \cap \mathcal{N}$ and $\mathcal{X}_m \cap \mathcal{N}$ such that it has at least $F + 1$ nodes outside the set. This was particularly challenging because in the original paper [30], the authors do not use all three conditions in the definition of $(F + 1, F + 1)$ robustness condition to informally prove the implication. They use only the third condition ($F + 1 \leq |\mathcal{X}_M^{F+1}| + |\mathcal{X}_m^{F+1}|$) to state the implication, while leaving it up on the readers to connect the missing dots with the first two conditions. For the implication to hold, all three conditions in the definition of $(F + 1, F + 1)$ -robustness should imply the existence of such a node since there is an *or* in the definition of $(F + 1, F + 1)$ -robustness connecting the three conditions. To prove the implication from the first two conditions, we need to first prove the existence of a normal node in the sets \mathcal{X}_M and \mathcal{X}_m for all $l \leq N$. This holds since the node i with value $M(t_\epsilon + l)$ will always be above the threshold $A_m - \epsilon_l$ because $M(t) \geq A_m, \forall t$ due to the existence of the limit A_m . Hence, $0 < |\mathcal{X}_M(t_\epsilon + l, \epsilon_l)|, \forall l \leq N$. Since the first condition of $(F + 1, F + 1)$ -robustness states that $|\mathcal{X}_M^{F+1}(t_\epsilon + l, \epsilon_l)| = |\mathcal{X}_M(t_\epsilon + l, \epsilon_l)|$, $0 < |\mathcal{X}_M^{F+1}(t_\epsilon + l, \epsilon_l)|$. Hence by definition of $\mathcal{X}_M^{F+1}(t_\epsilon + l, \epsilon_l)$, there exists a normal node in the set $X_M(t_\epsilon + l, \epsilon_l)$ such that it has at least $F + 1$ nodes outside $X_M(t_\epsilon + l, \epsilon_l)$. We prove this formally in Coq using the following lemma statement

```

Lemma X_m_normal_exists_at_j (t_eps 1 N: nat) (a A_m: R)(eps_0 eps:posreal)
(mal : nat → D → R) (init : D → R) (A: D → bool) (w: nat → D * D → R):
F_total_malicious mal init A w →
wts_well_behaved A mal init w →
(0 < F + 1 ≤ |D|) →
is_lim_seq [eta m mal init A w] A_m →
(0 < N) → (1 ≤ N) → (0 < a < 1) → (eps < a^N / (1 - a^N) * eps_0) →
∃ i:D, i ∈ (X_m_t_e_i (eps_j 1 eps_0 eps a) A_m (t_eps + 1) mal init A w) ∧
i ∈ Normal A.

```

By symmetry, we prove that $0 < |\mathcal{X}_m^{F+1}(t_\epsilon + l, \epsilon_l)|$. The other part that was not explicit from the paper proof in the original paper [30] was that the largest value that the node i uses at time step $t_\epsilon + l$ is $M(t_\epsilon + l)$, which is provided without proof. This was a challenge during our formalization. To formally prove this we had to split the neighbor set of i into two parts depending on their relative position with respect to i . While it is easy to bound the values of the

nodes positioned in the left side of i with $M(t_\epsilon + l)$ since the neighboring list is assumed to be sorted at the time of update and we have established this upper bound for any normal node from lemma 1, bounding the values for the nodes positioned in the right of the normal node i was not trivial. We proved this using a case analysis on the cardinality of the set $\mathcal{R}_i^>(t)$. In Coq, we formally prove this using the lemma statement `x_right_ineq_1` in Coq. We do not expand on this lemma here for brevity.

Another challenge during the formalization was using the bound of the neighboring node of i , $A_M - \epsilon_l$ in the update of the value of i at the next time step. We know that the neighbors outside the set $\mathcal{J}_i(t_\epsilon + l) \setminus \mathcal{X}_M(t_\epsilon + l, \epsilon_l)$ have value at most $A_M - \epsilon_l$. But to use these nodes in the update function, we need to show that these neighboring nodes are in the inclusive set of the normal node i minus the extremes, i.e, there exists a node in the intersection of the sets $\mathcal{J}_i(t_\epsilon + l)$ and the set s which contains nodes outside the set $\mathcal{J}_i(t_\epsilon + l) \setminus \mathcal{X}_M(t_\epsilon + l, \epsilon_l)$. We prove the existence of such a node using the following lemma statement in Coq

Lemma exists_in_intersection: $\forall (A B: \{\text{set } D\}) (s: \text{seq } D) (F: \text{nat}),$
 $|s| = (F+1)\%N \rightarrow (|B| \leq F)\%N \rightarrow$
 $\{\text{subset } s \leq A - B\} \rightarrow \exists x:D, x \in [\text{set } x \mid x \in s] \cap A.$

We instantiate the set A with $\mathcal{J}_i \setminus \mathcal{R}_i(t)$ and the set B with $\mathcal{R}_i^<(t)$. We know that by definition of the W-MSR algorithm, $|\mathcal{R}_i^<(t)| \leq F$. To use the lemma `exists_in_intersection`, we first had to prove that $s \subset (\mathcal{J}_i \setminus \mathcal{R}_i(t)) \cup \mathcal{R}_i^<(t)$. Applying the lemma `exists_in_intersection` then gives us a node k as a witness which lies in the intersection of the set s and $\mathcal{J}_i \setminus \mathcal{R}_i(t)$. We use this node to apply the bound $A_M - \epsilon_l$ in the proof of inequality 1 for $l \leq N$. All other nodes in the neighboring list of the normal node i minus extremes are shown to be bounded by $M(t)$.

To show that the inequality $\exists t, M(t) < A_M \vee A_m < m(t)$ holds, we need to prove that for every l such that $l \leq N$, the cardinality of the set \mathcal{X}_M decreases or the cardinality of the set \mathcal{X}_m decreases or both under the $(F+1, F+1)$ -robustness condition. This requires us proving the following lemma in Coq

Lemma sj_ind_var $(s1 s2: \text{nat} \rightarrow \text{nat}) (N: \text{nat}): (0 < N) \rightarrow (s1 \ 1 + s2 \ 1 < N) \rightarrow$
 $(\forall \ 1: \text{nat}, (0 < 1) \rightarrow (1 \leq N) \rightarrow (0 < s1 \ 1) \rightarrow (0 < s2 \ 1) \rightarrow$
 $(s1 \ 1 \leq s1 \ 1. - 1) \wedge (s2 \ 1 \leq s2 \ 1. - 1) \wedge ((s1 \ 1 < s1 \ 1. - 1) \vee (s2 \ 1 < s2 \ 1. - 1))) \rightarrow$
 $\exists T: \text{nat}, (T \leq N) \wedge (s1 \ T = 0 \vee s2 \ T = 0)$

We instantiate $s1$ and $s2$ with $\mathcal{X}_M(t_\epsilon + l, \epsilon_l)$ and $\mathcal{X}_m(t_\epsilon + l, \epsilon_l)$ respectively. We use the lemma `sj_ind_var` to arrive at a contradiction and complete the proof of the sufficiency.

3.3 Proof of necessity

Lemma 3. [30] *Consider a time-invariant network modeled by a digraph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ where each normal node updates its value according to the W-MSR algorithm with parameter F . Under the F -total malicious model, if resilient asymptotic consensus is achieved then the network is $(F+1, F+1)$ -robust.*

Necessity is a secondary, but still significant lemma. It tells us that there is no weaker condition than $(F + 1, F + 1)$ -robustness such that the normal nodes within the network reach asymptotic consensus. We now discuss an informal proof of Lemma 3. Note that the original paper [30] does not provide a clean proof of this lemma. For example, the original paper provides a sketch of the proof of Lemma 3 by contraposition, but does not provide a concrete counterexample to discharge the proof by contrapositive. The paper proof in [30] does not talk about construction of weights or the proof that these weights are not well-behaved under non- (r, s) -robustness. These issues were non-trivial and posed challenges in Coq, as will be explained in this section. We also highlight challenges in the construction of this counterexample and the proof of necessity in Coq, including an issue of mutual recursion in Coq. The issues with missing details in the original paper proof, which we had to develop explicitly, make the proof in this paper an original contribution.

Proof. We proceed by proving the contrapositive of necessity, that is: if the network is not $(F + 1, F + 1)$ robust then it does not achieve resilient asymptotic consensus. Assuming that the network is not $(F + 1, F + 1)$ -robust we know that there are non-empty sets $S_1, S_2 \subset \mathcal{V}$, such that $S_1 \cap S_2 = \emptyset$, $|\chi_{S_1}^{F+1}| \neq |S_1|$, $|\chi_{S_2}^{F+1}| \neq |S_2|$, and $|\chi_{S_1}^{F+1}| + |\chi_{S_2}^{F+1}| < F + 1$. It follows that $|\chi_{S_1}^{F+1}| < F + 1$, and $|\chi_{S_2}^{F+1}| < F + 1$. Also recall that $\chi_{S_1}^{F+1} \subseteq S_1$, and $\chi_{S_2}^{F+1} \subseteq S_2$. One way of interpreting this condition is that the number of nodes within S_1 and S_2 that can receive a lot of information from outside of their respective sets is less than $F + 1$ in total, and less than the number of nodes in each set respectively. We seek to construct a set of adversaries, initial values, malicious functions, and weights such that resilient asymptotic consensus is not achieved. In particular we seek to prove that there exists two normal nodes i, j such that $\lim_{t \rightarrow \infty} x_i(t) \neq \lim_{t \rightarrow \infty} x_j(t)$. We discuss the details of the proof in the Appendix D of the extended version [45].

Formalization in Coq: We formalize the lemma 3 in Coq as

Lemma necessity_proof:

```

nonempty_nontrivial_graph →
(¬ r_s_robustness (F + 1) (F + 1) →
  ¬ (∀ (A:D → bool) (mal:nat → D → R) (init:D → R) (w:nat → D * D → R),
    wts_well_behaved A mal init w →
    Resilient_asymptotic_consensus A mal init w)).

```

Formalization of **necessity_proof** exposed some inconsistencies in definitions in the original paper [30]. In particular, the paper defines those three conditions on weights, that we discussed in the Section 2.4, only for normal nodes. During our formalization, we found this to be restrictive. Those conditions on weights should hold for any node. The need for applying the conditions in the paper to the weights of adversary nodes, is that in order to ensure that a node $i \in \mathcal{A}$ is malicious, as defined in the paper, there must exist a time t such that the quantity $x_i(t + 1) \neq \sum_{j \in \mathcal{J}_i \setminus \mathcal{R}_i(t)} w_{ij}(t) x_j^i(t)$. In other words at some time the value emitted by a given node must not equal the value it would emit if it was

normal, but the sum is clearly undefined if the weights of an adversary node are undefined. Therefore, we relax the condition that the set of weights described in the paper only exists for normal nodes. Fortunately this does not create a problem as adversary nodes can update their values according to any function they wish, meaning that they do not have to use the described set of weights, or any weights at all, leaving their values unconstrained by this condition.

Another thing that was not explicit in the original paper [30] was the right placement of quantifiers. Formalizing the proof of necessity helped us identify the right placement of quantifiers and provide an accurate formal specification for the W-MSR algorithm. At the start of our formalization it was not evidently clear to us whether the paper meant to imply that:

$$(\forall (A:D \rightarrow \text{bool}) (\text{mal:nat} \rightarrow D \rightarrow R) (\text{init}:D \rightarrow R), \text{wts_well_behaved } A \text{ mal init} \rightarrow (\text{Resilient_asymptotic_consensus } A \text{ mal init} \leftrightarrow \text{r_s_robustness } (F + 1) (F + 1))).$$

or:

$$(\forall (A:D \rightarrow \text{bool}) (\text{mal:nat} \rightarrow D \rightarrow R) (\text{init}:D \rightarrow R), \text{wts_well_behaved } A \text{ mal init} \rightarrow \text{Resilient_asymptotic_consensus } A \text{ mal init}) \leftrightarrow \text{r_s_robustness } (F + 1) (F + 1).$$

In the first formula, the quantified values A , mal , init are not bound to the definition of resilient asymptotic consensus. Therefore, in the necessity proof, we cannot construct a counterexample by appropriate instantiation of A , mal and init , to discharge the proof by contradiction. In the second formula, the quantified values are bound to the definition of resilient asymptotic consensus, which allows us to construct the counterexample by propagating the negation through the quantified values. Essentially, the difference is between the formulae $(\forall X, P(X) \rightarrow Q(X))$ and $((\forall X. P(X)) \rightarrow (\forall X. Q(X)))$, where X represents the tuple $(A, \text{mal}, \text{init})$, and the first statement is stronger. Therefore, the former, stronger condition is not necessarily true in the necessity direction, while the weaker later condition is.

Another difficulty we encountered was defining the weights in such a way that $w_{ij}(t) = \frac{1}{|\mathcal{J}_i \setminus \mathcal{R}_i|}$. This is a result of Coq's sensitivity to ill-defined recursion. The issue arises because defining w_{ij} at time t requires knowing the value of x_i at time t , however, as we had defined x_i , it takes the set of weights it uses as a parameter, even though mathematically there is no issue since $x_i(t)$ only relies on the values of $x_j(t-1)$, and $w_{ij}(t-1)$. In order to solve this issue we defined a function which returns a pair of functions (x_i, w_{ij}) . In order to ensure that Coq could guess the parameter being recursed on we also had to add another parameter two_t which is initialized as $2 \cdot t$, and ensure that the pair $(x_i(t), w_{ij}(t))$ is returned when $\text{two}_t = 2 \cdot t$, and $(x_{t+1}, w_{ij}(t))$ is returned when $\text{two}_t = (2 \cdot t) + 1$.

3.4 Formal proof of the main theorem

We state the main theorem statement 1 in Coq as:

Theorem F_total_consensus :
 $\text{nonempty_nontrivial_graph} \rightarrow$

$(0 < F+1 \leq |D|)\%N \rightarrow$
 $(\forall (A:D \rightarrow \text{bool}) (\text{mal}:\text{nat} \rightarrow D \rightarrow R) (\text{init}:D \rightarrow R) (\text{w}:\text{nat} \rightarrow D * D \rightarrow R),$
 $\text{wts_well_behaved } A \text{ mal init w} \rightarrow$
 $\text{Resilient_asymptotic_consensus } A \text{ mal init w} \leftrightarrow \text{r_s_robustness } (F+1) (F+1).$

We close the proof of `F_total_consensus` by splitting the theorem into sufficiency and necessity sub-proofs and applying the lemmas `sufficiency_proof` and `necessity_proof`. The only detail worth noting is that `necessity_proof` relies on the decidability of `r_s_robustness`, which we need the axiom of the excluded middle to conclude.

4 Related Work

Recently there has been a growing interest in the formalization of distributed systems and control theory, using both automated and interactive verification approaches.

Some notable works in the area of automated verification use model checking, temporal logic, and reachability techniques. For instance, Cimatti et al. [11] have used model checking techniques to formally verify the implementation of a part of safety logic for railway interlocking system. Schrer et al. [43] extended the JavaPathFinder [24] model checker to support modeling of a real-time scheduler and physical system that are defined by differential equations. They verify the safety and liveness properties of a control system, and also verify the programming errors. Besides model checking, temporal logic based techniques have been applied to control synthesis [40], robust model predictive control [14] and automatic verification of sequential control systems [35]. Other approaches for verifying safety use reachability methods like flow pipe approximations [10], zonotope approximation algorithms [19, 28, 2], and ellipsoidal calculus [4].

There has also been significant work in the formalization of control theory using interactive theorem provers [39, 1, 38]. In the area of formalization of stability analysis for control theory, Cyril Cohen and Damien Rouhling formalized the LaSalle's principle in Coq [12]. Stability is important for the control of dynamical systems since it guarantees that trajectories of dynamical systems like cars and airplanes, are bounded. Chan et al. [7] formalize safety properties like Lyapunov stability and exponential stability of cyber-physical systems, in Coq. In [39], Damien Rouhling formalized the soundness of a control function [32] for an inverted pendulum. Some works have also emerged in the area of signal processing for controls. Gallois-Wang et al. [17] formalized some error analysis theorems about digital filters in Coq. Araiza-Illan et al. [3] formally verified high level properties of control systems such stability, feedback gain, or robustness using the Why3 tool [15]. Rashid et al. [38] formalized the transform methods in HOL-Light [22]. Transform methods are used in signal processing and controls to switch between the time domain and the frequency domains for design and analysis of control systems. A few works have emerged in the area of formalization of the feedback control theory to guarantee robustness of control systems. Jasim and Veres et al [26] proved one of the most fundamental and general

result of nonlinear feedback system - the *Small-gain theorem (SGT)*, formally using Isabelle/HOL [37]. Hasan et al [23] formalized the theoretical foundations of feedback controls in HOL Light. Another notable work in the formalization of control systems is the formalization of safety properties of robot manipulators by Affeldt et al. [1].

Most of the above works deal with the problem of formalizing the theoretic foundations of control theory – stability analysis, transform methods, filtering algorithms for signal processing, feedback control design. But, to our knowledge, none of these works tackles the problem of consensus in a formal setting. Given that consensus is a quantity of interest in distributed control applications, our work on the formalization of the W-MSR algorithm, is a first step towards formally verified distributed control systems.

5 Conclusion

In this work, we formalize a consensus algorithm [30] for distributed controls in Coq. We formally prove the necessary and sufficient conditions for a set of normal nodes in the network to achieve asymptotic consensus in the presence of a fix bound of malicious nodes in the network. During the process of formalization we discover several areas where the proof in the original paper is imprecise, especially when defining the lemma statements of sufficiency and necessity. In particular, the order of quantifiers on some variables was unclear, and we had to spend time clarifying their order. We also prove a stronger version of the sufficiency condition than the original theorem requires. This is done to ensure that the conditions in both directions of the double implication holds. The definitions and lemmas we formalize in this paper can be used for verifying consensus for other threat models described in the original paper [30]. Overall our work is a first of its kind to provide formal specifications of a consensus algorithm in distributed controls. The total length of Coq proofs is about 11 thousand lines of code. It took us 6 person months for the entire formalization.

A possible future direction of work is to verify the implementation of the algorithm. The proof of this algorithm in the original paper [30], and our formalization assume that all computations are in the real field. However, an actual implementation would need to use finite precision arithmetic. It would therefore be interesting to study the effect of finite precision on the robustness of this algorithm. It would also be interesting to formalize the algorithm for time-variant networks in which the edge relation between the nodes can change with time. Possible use cases for such network model are drone swarms for military and rescue operations, in which each drone in the network could be expected to dynamically change the flow of information from its neighbors.

Acknowledgments: This research was funded in part by NSF grant CCF-2219997.

References

1. Affeldt, R., Cohen, C.: Formal foundations of 3d geometry to model robot manipulators. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. pp. 30–42 (2017)
2. Althoff, M., Krogh, B.H.: Zonotope bundles for the efficient computation of reachable sets. In: *2011 50th IEEE conference on decision and control and European control conference*. pp. 6814–6821. IEEE (2011)
3. Araiza-Illan, D., Eder, K., Richards, A.: Formal verification of control systems’ properties with theorem proving. In: *2014 UKACC International Conference on Control (CONTROL)*. pp. 244–249. IEEE (2014)
4. Botchkarev, O., Tripakis, S.: Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In: *International Workshop on Hybrid Systems: Computation and Control*. pp. 73–88. Springer (2000)
5. Carr, H., Jenkins, C., Moir, M., Miraldo, V.C., Silva, L.: Towards formal verification of hotstuff-based byzantine fault tolerant consensus in agda. In: *NASA Formal Methods Symposium*. pp. 616–635. Springer (2022)
6. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: *OSDI*. vol. 99, pp. 173–186 (1999)
7. Chan, M., Ricketts, D., Lerner, S., Malecha, G.: Formal verification of stability properties of cyber-physical systems. *Proc. CoqPL* (2016)
8. Charron-Bost, B., Merz, S.: Formal verification of a consensus algorithm in the heard-of model. *Int. J. Softw. Informatics* **3**(2-3), 273–303 (2009)
9. Charron-Bost, B., Merz, S.: Formal Verification of a Consensus Algorithm in the Heard-Of Model. *International Journal of Software and Informatics (IJSI)* **3**(2-3), 273–303 (2009), <https://inria.hal.science/inria-00426388>
10. Chutinan, A., Krogh, B.H.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: *International workshop on hybrid systems: computation and control*. pp. 76–90. Springer (1999)
11. Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P.: Formal verification of a railway interlocking system using model checking. *Formal aspects of computing* **10**(4), 361–380 (1998)
12. Cohen, C., Rouhling, D.: A formal proof in coq of lasalle’s invariance principle. In: *International Conference on Interactive Theorem Proving*. pp. 148–163. Springer (2017)
13. Doczkal, C., Pous, D.: Graph theory in coq: Minors, treewidth, and isomorphisms. *Journal of Automated Reasoning* **64**(5), 795–825 (2020)
14. Farahani, S.S., Raman, V., Murray, R.M.: Robust model predictive control for signal temporal logic synthesis. *IFAC-PapersOnLine* **48**(27), 323–328 (2015)
15. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: *European symposium on programming*. pp. 125–128. Springer (2013)
16. Függer, M., Nowak, T., Schwarz, M.: Tight bounds for asymptotic and approximate consensus. *Journal of the ACM (JACM)* **68**(6), 1–35 (2021)
17. Gallois-Wong, D., Boldo, S., Hilaire, T.: A coq formalization of digital filters. In: *International Conference on Intelligent Computer Mathematics*. pp. 87–103. Springer (2018)
18. Gao, S., Zhan, B., Liu, D., Sun, X., Zhi, Y., Jansen, D.N., Zhang, L.: Formal verification of consensus in the taurus distributed database. In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*. pp. 741–751. Springer (2021)

19. Girard, A., Guernic, C.L.: Zonotope/hyperplane intersection for hybrid systems reachability analysis. In: International Workshop on Hybrid Systems: Computation and Control. pp. 215–228. Springer (2008)
20. Goel, A., Sakallah, K.: On symmetry and quantification: A new approach to verify distributed protocols. In: NASA Formal Methods Symposium. pp. 131–150. Springer (2021)
21. Hance, T., Heule, M., Martins, R., Parno, B.: Finding invariants of distributed systems: It's a small (enough) world after all. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). pp. 115–131 (2021)
22. Harrison, J.: Hol light: A tutorial introduction. In: International Conference on Formal Methods in Computer-Aided Design. pp. 265–269. Springer (1996)
23. Hasan, O., Ahmad, M.: Formal analysis of steady state errors in feedback control systems using hol-light. In: 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1423–1426. IEEE (2013)
24. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer **2**(4), 366–381 (2000)
25. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 1–17 (2015)
26. Jasim, O.A., Veres, S.M.: Towards formal proofs of feedback control theory. In: 2017 21st International Conference on System Theory, Control and Computing (ICSTCC). pp. 43–48. IEEE (2017)
27. Kieckhafer, R.M., Azadmanesh, M.H.: Reaching approximate agreement with mixed-mode faults. IEEE Transactions on Parallel and Distributed Systems **5**(1), 53–63 (1994)
28. Kochdumper, N., Althoff, M.: Sparse polynomial zonotopes: A novel set representation for reachability analysis. IEEE Transactions on Automatic Control **66**(9), 4043–4058 (2020)
29. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
30. LeBlanc, H.J., Zhang, H., Koutsoukos, X., Sundaram, S.: Resilient asymptotic consensus in robust networks. IEEE Journal on Selected Areas in Communications **31**(4), 766–781 (2013)
31. Losa, G., Dodds, M.: On the formal verification of the stellar consensus protocol. In: 2nd Workshop on Formal Methods for Blockchains (FMBC 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
32. Lozano, R., Fantoni, I., Block, D.J.: Stabilization of the inverted pendulum around its homoclinic orbit. Systems & control letters **40**(3), 197–204 (2000)
33. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: incremental inference of inductive invariants for verification of distributed protocols. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 370–384 (2019)
34. Mesbahi, M., Egerstedt, M.: Graph theoretic methods in multiagent networks. Princeton University Press (2010)
35. Moon, I., Powers, G.J., Burch, J.R., Clarke, E.M.: Automatic verification of sequential control systems using temporal logic. AIChE Journal **38**(1), 67–75 (1992)
36. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). pp. 305–319 (2014)
37. Paulson, L.C.: Isabelle: A generic theorem prover. Springer (1994)

38. Rashid, A., Hasan, O.: Formalization of transform methods using hol light. In: International Conference on Intelligent Computer Mathematics. pp. 319–332. Springer (2017)
39. Rouhling, D.: A formal proof in coq of a control function for the inverted pendulum. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 28–41 (2018)
40. Sadraddini, S., Belta, C.: Robust temporal logic model predictive control. In: 2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton). pp. 772–779. IEEE (2015)
41. Saldana, D., Prorok, A., Sundaram, S., Campos, M.F., Kumar, V.: Resilient consensus for time-varying networks of dynamic agents. In: 2017 American control conference (ACC). pp. 252–258. IEEE (2017)
42. Saulnier, K., Saldana, D., Prorok, A., Pappas, G.J., Kumar, V.: Resilient flocking for mobile robot teams. *IEEE Robotics and Automation letters* **2**(2), 1039–1046 (2017)
43. Scherer, S., Lerda, F., Clarke, E.M.: Model checking of robotic control systems (2005)
44. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* **22**(4), 299–319 (dec 1990). <https://doi.org/10.1145/98163.98167>, <https://doi.org/10.1145/98163.98167>
45. Tekriwal, M., Tachna-Fram, A., Jeannin, J.B., Kapritsos, M., Panagou, D.: Formally verified asymptotic consensus in robust networks (extended version). *arXiv preprint arXiv:2202.13833* (2022)
46. Usevitch, J., Garg, K., Panagou, D.: Finite-time resilient formation control with bounded inputs. In: 2018 IEEE Conference on Decision and Control (CDC). pp. 2567–2574. IEEE (2018). <https://doi.org/10.1109/CDC.2018.8619697>
47. Van Renesse, R., Altinbukan, D.: Paxos made moderately complex. *ACM Computing Surveys (CSUR)* **47**(3), 1–36 (2015)
48. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 357–368 (2015)
49. Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., Ryan, G.: DistAI: Data-driven automated invariant learning for distributed protocols. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). pp. 405–421 (2021)
50. Zhang, H., Sundaram, S.: Robustness of information diffusion algorithms to locally bounded adversaries. In: 2012 American Control Conference (ACC). pp. 5855–5861. IEEE (2012)

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

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





Formally Verifying an Efficient Sorter

Bernhard Beckert, Peter Sanders, Mattias Ulbrich ^(✉), Julian Wiesler,
and Sascha Witt

Karlsruhe Institute of Technology, Karlsruhe, Germany
{beckert,sanders,ulbrich,sascha.witt}@kit.edu

Abstract. In this experience report, we present the complete formal verification of a Java implementation of inplace superscalar sample sort (ips⁴o) using the KeY program verification system. As ips⁴o is one of the fastest general purpose sorting algorithms, this is an important step towards a collection of basic toolbox components that are both provably correct and highly efficient. At the same time, it is an important case study of how careful, highly efficient implementations of complicated algorithms can be formally verified directly. We provide an analysis of which features of the KeY system and its verification calculus are instrumental in enabling algorithm verification without any compromise on algorithm efficiency.

1 Introduction

The core task of computer scientists can be seen as writing correct and efficient computer programs. However, although both correctness and efficiency have been intensively studied, there is comparably little work on fully combining both features. We would like *formally verified* code that is *efficient on modern machines*. We believe that a library of verified high-performance implementations of the basic toolbox of most frequently used algorithms and data structures is a crucial step towards this goal: often, these components take a considerable part of the overall computation time, and they have a simple specification which allows reusing their verified functionality in a large number of programs. Since the remaining code may be simpler from an algorithmic point of view, verifying such programs could thus be considerably simplified.

To make progress in this direction, we perform a case study on sorting, which is one of the most frequently used basic toolbox algorithms. For example, a recent study identified hundreds of performance relevant sorting calls in Google's central software depot [36]. Taking correctness of even standard library routines for granted is also not an option. For example, during a verification attempt of the built-in sorting routine of the OpenJDK TimSort routine, researchers were able to detect a bug, using the KeY verifier [11].

Although some sorters have been formally verified [12,4,20], it turns out that these do not achieve state-of-the-art performance because only rather simple combinations and variants of quicksort, mergesort, or heapsort have been used

that lack cache efficiency when applied to large data sets and have performance bottlenecks that limit instruction parallelism. The best available sorters are considerably more complex (≈ 1000 lines of code) and even more likely to contain bugs when not formally verified. Moreover, previous verifications do not prove all required properties or they operate only on an abstraction of the code, which makes it difficult to relate to highly tuned implementations.

For our verification of a state-of-the-art sorter, we consider `ips4o` (**i**n-**p**lace **s**uper **s**calar **s**ample **s**ort) [2]. Sample sort [10] generalises quicksort by partitioning the data into many pieces in a single pass over the data, which makes it more cache efficient (indeed I/O-optimal up to lower order terms). Additionally, `ips4o` works in-place (an important requirement for standard libraries and large inputs), avoids branch mispredictions, and allows high instruction parallelism by reducing data dependencies in the innermost loops. The algorithm also has an efficient parallelisation and parts of it can be used for fast integer sorting [2,36]. Extensive experiments indicate that a C++ implementation of `ips4o` considerably outperforms quicksort, mergesort and heapsort on large inputs and is several times faster than adaptive sorters such as TimSort on inputs that are not already almost sorted [2]. Our experiments in Sec. 5 indicate that the verified Java implementation is 1.3 to 1.8 times faster than the standard library sorter of OpenJDK 20 for large inputs on three different architectures.

We use the Java Modeling Language (JML) [22] to directly specify the efficient Java implementation of sequential `ips4o`. We obtain a largely automated proof using the KeY theorem prover [1] in part aided by external theory solvers (in particular Z3 [33]) and KeY’s support for interactively guiding the proof construction process. This yields a full functional correctness proof of the full Java implementation of `ips4o` showing, for all possible inputs, *sortedness*, the *permutation property*, *exception safety*, *memory safety*, *termination*, and *absence of arithmetic overflows*. The complete 8-line specification of the toplevel sorting method can be seen in Fig. 1.

The verified code is available for download¹ and can easily be used in real-world Java applications (through the maven packaging mechanism). It spans over 900 lines of Java code with the main properties specified on 8 lines of JML, annotated with some 2500 lines of JML auxiliary annotations for prover guidance. The project required a total of 1 million proof steps (of which 4000 were performed manually) on 179 proof obligations (with one or more proof obligation per Java method). The project required about 4 person months.

The verification revealed a subtle bug in the original version, where the algorithm would not terminate if presented with an array containing the same single value many times.² This flaw was subsequently fixed. Moreover, the formal verification revealed that the code could be simplified at one point.

This case study demonstrates that competitive code hand-optimised for the application on modern processors can be deductively verified within a reason-

¹ at the github repository <https://github.com/KeYProject/ips4o-verify>

² The bug was latently present in the original C++-code also. However, it cannot occur when the default parameter values are used in C++.

able time frame. It resulted from a fruitful collaboration of experts in program verification and experts in algorithm engineering. An extended version of this paper [3] is available containing more in-depth information about the specification and verification.

2 Background

2.1 Formal Specification with the Java Modeling Language

The Java Modeling Language (JML) [22] is a behavioural interface specification language [15] following the paradigm of design-by-contract [29]. JML is the *de-facto* standard for the formal specification of Java programs. The main artefact of JML specifications are method contracts comprised of preconditions (specified via **requires** clauses), postconditions (**ensures**) and a frame condition (**assignable**) which describes the set of heap locations to which a method invocation is allowed to write. A contract specifies that, if a method starts in a state satisfying its preconditions, then it must terminate and the postcondition must be satisfied in the post-state of the method invocation. Additionally, any modified heap location already allocated at invocation time must lie within the specified assignable clause. Termination witnesses (**measured.by** clauses) are used to reason about the termination of recursive methods. Java loops can be annotated with invariants (**loop_invariant**), which must be true whenever the loop condition is evaluated, termination witnesses (**decreases**), and frame conditions (**assignable**) that limit the heap locations the loop body may modify. Loop specifications and method contracts of internal methods allow one to conduct proofs modularly and inductively.

Expressions in JML are a superset of side-effect-free Java expressions. In particular, JML allows the use of field references and the invocation of pure methods in specifications. JML-specific syntax includes first-order quantifiers (**\forall** and **\exists**) and generalised quantifiers. One generalised quantifier is the construct (**\num_of** T x ; φ) which evaluates to the number of elements of type T that satisfy the condition φ (if that number is finite). (**\sum** T x ; φ ; e) sums the expression e over all values of type T satisfying φ . Quantifiers in JML support range predicates to constrain the bound variable; the expression (**\forall** T x ; φ ; ψ) is hence equivalent to (**\forall** T x ; $\varphi \implies \psi$).

JML specifications are annotated in the Java source code directly and enclosed in special comments beginning with **/*@** or **/**@** to allow them to be compiled by a standard Java compiler. JML supports the definition of verification-only (model and ghost) entities within JML comments that are only visible at verification time and do not influence runtime behaviour (see also Sec. 4.1).

Fig. 1 shows the specification of the top-level **sort** method as an example. Since that JML contract is labelled **normal_behaviour**, it requires (in addition to satisfying the pre-post contract) that the method does not terminate abruptly by throwing an exception.

```

1  /*@ public normal_behaviour
2     @   requires v.length <= MAX_LEN;
3     @   ensures seqPerm(array2seq(v), \old(array2seq(v)));
4     @   ensures (\forallall int i; 0 <= i < v.length-1; v[i] <= v[i+1]);
5     @   assignable v[*];
6  @*/
7  public static void sort(int[] v) { ... }

```

Fig. 1: Specification of the sorting entry method specifying that after the method call, the array `values` contains a permutation of the input values (line 3) and is sorted (quantified expression in line 4). Only entries in the array are modified in the process (line 5).

2.2 Deductive Verification with the KeY System

The KeY verification tool [1] is a deductive theorem prover which can be used to verify Java programs against JML specifications. KeY translates JML specifications into proof obligations formalised in the dynamic logic [13] variant JavaDL, in which Java program fragments can occur within formulas. The JavaDL formula $\varphi \rightarrow \langle \text{o.m}() \rangle \psi$ is similar to the total Hoare triple $[\varphi] \text{o.m}(); [\psi]$, with both stating that the method invocation `o.m()` terminates in a state satisfying ψ if started in a state satisfying φ . Proofs in KeY are conducted by applying inference rules in a sequent calculus. Using a set of inference rules for Java statements, the Java code $(\langle \text{o.m}() \rangle \psi)$ in the above statement) is symbolically executed such that the approach yields the weakest precondition for `o.m()` and ψ as a formula in first-order predicate logic. KeY can settle many proof obligations automatically, but also allows interactive rule application and invocation of external provers like satisfiability modulo theories (SMT) solvers.

3 Our Java Implementation of `ips4o`

3.1 The Algorithm

In-place (parallel) super scalar sample **sort** (`ips4o`), is a state-of-the-art general sorting algorithm [2]. Sample sorting can be seen as a generalisation of quick sort, where instead of choosing a single pivot to partition elements into two parts, we choose a sorted sequence of $k - 1$ *splitters* which define k *buckets* consisting of the elements lying between adjacent splitters. One advantage of this is the reduced recursion depth and the resulting better cache efficiency. “Super-scalar” refers to enabling instruction parallelism by avoiding branches and reducing data dependencies while classifying elements into buckets. “In-place” means that the algorithm needs only logarithmic space in addition to the input. Although `ips4o` has a parallel version, this work is concerned with the sequential case.

The algorithm works by recursively partitioning the input into buckets; when the sub-problems are small enough, they are sorted using insertion sort. The maximum number of buckets k_{\max} and the base-case size, i.e., the maximum

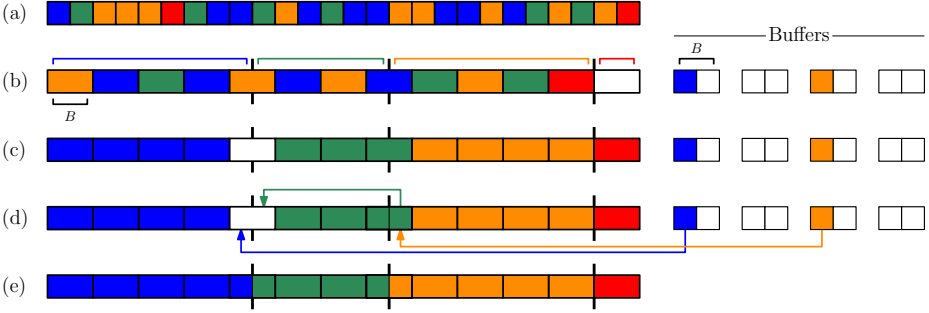


Fig. 2: Overview of all steps of *ips⁴o*: (a) input with elements classifying as the four classes blue, green, orange and red, (b) After classification ($B = 2$); bucket sizes are indicated by brackets and white elements are empty, (c) after permutation, (d) the operations done by the cleanup step, (e) partitioned output.

problem size for insertion sort, are configuration parameters. In our implementation, we chose $k_{\max} = 256$ and base-case size 128 experimentally. Partitioning consists of four steps: Sampling, classification, permutation, and cleanup.

Sampling. This step finds the splitters as equally spaced elements from a (recursively) sorted random sample of the current subproblem. There are special cases to handle small or skewed inputs. These are fully handled in our proof, but to simplify the exposition, we will assume in this summary that $k = k_{\max}$ distinct³ splitters are found this way.

Classification. The goal of the classification step is two-fold: (1) to assign each element to one of the k buckets defined by the splitters, and (2) to pre-sort elements into fixed-size blocks such that all elements in a block belong to the same bucket. To find the right bucket for each element, the largest splitter element smaller than that element must be identified. A number of algorithm engineering optimisations make the classification efficient: it is implemented using an implicit perfect binary search tree with logarithmic lookup complexity. Moreover, the tree data structure also supports an implementation without branching statements and unrolled loops that eliminates branch mispredictions and facilitates high instruction parallelism and the use of SIMD instruction. We will come back to this classification tree implementation in Sect. 4.2 where we discuss how this efficiency choice was dealt with in the formal proof.

After classification is done, the input array consists of blocks in which all elements belong to the same bucket, followed by some empty space, with the remaining elements still remaining in the (partially filled) buffers. The block size B is chosen experimentally to be 1 KiB. Fig. 2.b shows the output of this step.

Permutation. By now, it is known how many elements are in each bucket, and therefore where in the array each bucket begins and ends after partitioning is done. The objective of the permutation step is to rearrange the blocks so that

³ If equal splitters do appear, duplicates are removed and *equality buckets* are used that do not require recursive sorting. Details can be found in the extended version [3].

each block starts in the correct bucket. Then, if the block is not already correctly placed, it is moved to its bucket, possibly displacing another (incorrectly placed) block, which is then similarly moved. Refer to Fig. 2.c for the state of the input array after this step.

Cleanup. In general, bucket boundaries will not coincide with block boundaries. Since the permutation step works on block granularity, there may be overlap where elements spill into an adjacent bucket. These elements are corrected in the cleanup step. In addition, the remaining elements in the buffers from the classification step are written back into the input array. Fig. 2.d shows an example of the steps performed during cleanup.

3.2 Algorithm Engineering for Java

While the original implementation of `ips4o` was written in C++, the verification target of this case study is a translation by one of the authors of the original code to Java. No performance-relevant compromises were made, e.g., to achieve easier verification. We started with a Java implementation as close as possible to the C++ implementation. We then performed profiling-driven tuning. Adjusting configuration parameters improved performance by 12%. The only algorithmically significant change resulting from tuning is when small sub-problems are sorted. In the C++ implementation this is done during cleanup in order to improve cache locality. In Java it turned out to be better to remove this special case, i.e., to sort all sub-problems in the recursion step. This improved performance by a further 4%.

4 Specification and Verification

In this case study, the following properties of the Java `ips4o` implementation have been specified and successfully verified:

Sorting Property: The array is sorted after the method invocation.

Permutation Property: The content of the input array after sorting is a permutation of the initial content.

Exception Safety: No uncaught exceptions are thrown.

Memory Safety: The implementation does not modify any previously allocated memory location except the entries of the input array.

Termination: Every method invocation terminates.

Absence of Overflows: During the execution of the method, no integer operation will overflow or underflow.

We assume that no out-of-memory or stack-overflow errors can ever occur at runtime. Since the algorithm is in-place, and the recursion depth is in $\mathcal{O}(\log n)$, this is a reasonable assumption to make.

Fig. 1 shows the JML specification of the entry method `sort` of the `ips4o` implementation, i.e., the top-level requirements specification of the sorting algorithm. The annotation `normal_behaviour` in line 1 specifies exception safety (i.e.

the absence of both explicitly and implicitly thrown uncaught exceptions). Memory safety is required by the framing condition in line 5. The permutation and sorting property are formulated as postconditions in lines 3 resp. 4. Termination is a default specification case with JML (unless explicitly specified otherwise). The absence of overflows is not specified in JML, but is an option that can be switched on in KeY. The precondition in line 2 of the method contract ensures that there are no overflows and is of little practical restriction since it is very close to the maximum integer value ($\text{MAX_LEN} = 2^{31} - 256$).

The implementation of Java `ips4o` comprises 900 lines of code, annotated with 2500 lines in JML. Besides the requirement specification, this comprises auxiliary specifications such as method contracts for (sub-)methods, class and loop invariants, function or predicate definitions and lemmata. We will focus on selected specification items and emphasise the algorithm’s classification step since it has sophisticated, interesting loop invariants that are at the same time comprehensible, exemplifying the techniques we were using.

4.1 Enabling KeY Features

A few advanced features of KeY were essential for completing the proof. They are needed to *abstract* from sophisticated algorithmic concepts and to *decompose* larger proofs into more manageable units.

We followed a mostly *autoactive* program verification approach [25] with as much *automation* as possible while supporting *interactive* prover guidance in form of source code annotations (e.g. assertions). This concept has been widely adapted throughout the program verification community [35,31,24,9]. Most program verification tools only allow guidance by source code annotations. However, the KeY theorem prover also supports an interactive proof mode in which inference rules can be applied manually – and we resorted also to this way of proof construction where needed.

Model methods. Due to the scale of the project, it was useful to encapsulate important properties of the data structures into named abstract predicates or functions. The vehicle to formulate such abstraction in JML are *model methods* [32], which are side-effect free (*pure*) methods defined within JML annotations. For `ips4o`, around 100 different model methods were used.

The benefits of using model methods are two-fold: (1) They structure and decompose specifications making them more comprehensible and (2) they simplify resp. enable automated verification by abstraction of the proof state. An example for a widely used (50 occurrences) model method is shown in Fig. 3.

Ghost fields and variables provide further abstractions from the memory state by defining verification-only memory locations. In the present case study, all Java classes except simple pure data containers required at least one ghost field. Sec. 4.2 reports a challenge were ghost variables and ghost code (i.e. assignments to ghost variables) made verification possible in the first place.

Assertions are the main proof-guidance tool in autoactive verification as they provide means to formulate intermediate proof targets that the automation can discharge more easily and that thus may provide a deductive chain


```

1  /*@ public model_behaviour
2    @ accessible values[begin..end - 1];
3    @ static model int countElement(int[] values, int begin, int end, int e) {
4    @   return (\num_of int i; begin <= i < end; values[i] == e); } */

```

Fig. 3: Model method that counts the occurrences of the integer `element` in the index range `begin, ..., end - 1`. The `accessible` clause specifies that the model method may only read the `values` between `begin` and `end-1` (inclusively).

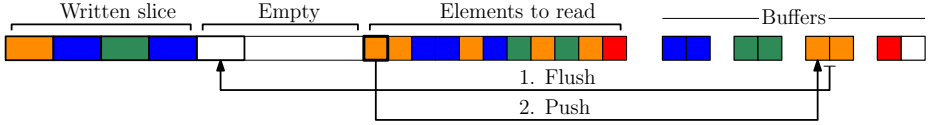


Fig. 4: Intermediate state of the classification step after processing some elements. The first element to be read is being pushed to the orange buffer which gets flushed beforehand.

completing the proof. This corresponds to making case distinctions or to introducing intermediate goals in a manual proof. In the present case study, assertions avoided many tedious interactive proof steps as the annotations in the source code guide the proof search such that it now runs automatically.

Block contracts. Much like method contracts, block contracts abstract from details in control flow and implementation details of a Java code block they annotate (similar to a method contract). Block contracts can decompose large and complex method implementation and allow one to focus on the relevant effects of individual components (i.e., code blocks) formalised in the postconditions of the block contracts.

4.2 Central Ideas Used in the Proofs of the Steps of `ips4`

In this section we zoom in on a few central concepts from the proofs of the algorithm. We mainly focus on the classification step which (1) establishes the most relevant invariants of the recursion step, and (2) showcases a particular proof technique related to the verification of the efficient algorithm implementation used in this case study.

Relevant Invariants. During classification, the algorithm rearranges the input elements into blocks (of a given size B) such that all elements in a block are classified into the same bucket. Furthermore, it counts the elements in each bucket. Fig. 4 shows an intermediate state of the classification step. It is checked to which bucket the next element belongs, that bucket's buffer is *flushed* if needed, and then the element is *pushed* to the buffer according to its classification. This is done in batches of m elements at once such that the classification can take advantage of batched queries (that allow the CPU to apply instruction parallelism).

```

1  /*@ loop_invariant begin <= i <= end && begin <= write <= i;
2  loop_invariant (\forallall int b; 0 <= b < num_buckets; (\forallall int i;      // (1)
3    b * BUFFER_SIZE <= i < b * BUFFER_SIZE + buffers.lengths[b];
4    classOf(buffers.buffer[i]) == b));
5  loop_invariant (\forallall int block; 0 <= block < (end-begin)/BUFFER_SIZE; // (2)
6    (\exists int b; 0 <= b < num_buckets; (\forallall int i;
7      begin + block * BUFFER_SIZE <= i < begin + (block+1)*BUFFER_SIZE;
8      classOf(values[i]) == b));
9  loop_invariant (\forallall int element;                                     // (3)
10    \old(countElement(values, begin, begin, begin, end, buffers, element)) ==
11    countElement(values, begin, write, i, end, buffers, element)
12  loop_invariant (\forallall int b; 0 <= b < num_buckets; bucket_counts[b] == // (4)
13    (\num_of int i; begin <= i < write; classOf(values[i]) == b));
14  loop_invariant write - begin == (\sum int b;                               // (5)
15    0 <= b < num_buckets; bucket_counts[b]);
16  loop_invariant (\forallall int b; 0 <= b < num_buckets;                     // (6)
17    isValidBufferLen(buffers.lengths[b], bucket_counts[b]));
18  loop_invariant buffers.count() == i - write;                               // (7a)
19  loop_invariant (i - begin) loop_invariant (write - begin)

```

Fig. 5: Specification of the classification loop. `begin` and `end` are the boundaries of the slice that is being processed, `i` is the offset of the next element that will be classified, `write` is the end offset of the written slice. The array `bucket_counts` contains the element count for each bucket.

After classifying all elements, the count of all elements in each bucket's buffer is added to get the full element count for each bucket. We define the *written slice* to be the elements that were already flushed to the input array.

To exemplify the nature of the specification used in this case study, we discuss the inductive loop invariants of the classification loop which allowed us to close the proof for this step. Fig. 5 shows the corresponding JML annotations⁴.

1. The buffers contain only bucket elements of their respective bucket.
2. The written slice is made up of blocks of size B where each block contains only elements of exactly one bucket.
3. The permutation property is maintained.
4. The per bucket element counts are exactly the number of elements of the corresponding bucket in the written slice.
5. The sum of all per bucket element counts equals the size of the written slice.
6. The buffer size of each bucket is valid.
7. The spacings are well formed:
 - (a) The total element count in all buffers equals the length of the free slice.
 - (b) The start offset of the current batch is a multiple of m .
 - (c) The length of the written slice is a multiple of B .

Invariants 1 and 2 straightforwardly encode the block structure during classification from the abstract algorithm. They are also needed as preconditions

⁴ In the actual implementation, the invariants are grouped in several model methods.

for the following partitioning step. The permutation invariant 3 ensures that no elements are lost during classification by stating that the original array content is a permutation of the union of all elements not yet handled, the written slice and the union of all buffers. Invariants 4 and 5 are needed to show that the bucket element counts are correct and to show that all elements of the input will have been taken into account eventually. These invariants were engineered by translating the ideas from the abstract algorithm into the Java situation. The remaining two invariants were discovered later in the verification process: The validity invariant 6 was only discovered during the proof of the cleanup step (where it becomes relevant). A buffer is called *valid*, if (1) the number of elements written back during classification is a multiple of the block size B and (2) empty buffers are only allowed when nothing has yet been written back. Invariant 7 was discovered last by inspecting the open proof goals of failed attempts, and is mostly needed to show that write operations to the heap remain in bounds.

Invariant 5, while in principle derivable from the other invariants, simplifies the proof that the sum of all bucket element counts is the size of the input after termination. Adding it as a redundant loop invariant avoids having to prove the same statement repeatedly using the other invariants.

When flushing a buffer, the algorithm must not overwrite the batch that it is currently processing nor the elements that were not processed yet. This property is captured in invariant 7. First and foremost, 7a ensures that there is enough space to write a whole buffer if a buffer is full. When pushing the elements of the current batch to their buckets, the algorithm makes sure that the start of the batch will never be overwritten. However, this was not provable from the scope of this loop: For example, let there be B total elements in all buffers, all of which are in the buffer of some bucket b when we are trying to push the second element of a batch to b 's buffer. A flush may then happen before the push which would illegally overwrite the first element of the batch. This case is shown to be impossible by adding invariants 7b and 7c. In general, this holds for any values where B is a multiple of the batch size m .

Classification Search Tree. As mentioned in Sec. 3, classification employs an implicit binary search-tree data structure to find the bucket to which an element belongs. This is a complete binary tree where the root of a subtree stores the median of the splitters belonging to the subtree. The splitters are stored in an array with the root at index one. The children of the node stored at index i are stored at indices $2i$ and $2i + 1$. Fig. 6 shows the branch-free loop to compute the bucket $c(e)$ for an element e .

It was difficult to verify this routine with hard to find loop invariants. On the other hand, an implementation using binary search on a linearly sorted array would have been easier to verify; but without the benefits of branch-freedom. Hence, this optimisation is an example where algorithm engineering decisions make verification more complicated. Our solution to the problem was to implement the binary search algorithm on the array of indices in parallel next to the efficient tree search by means of ghost variables and ghost code. A set of *coupling invariants* set the variables of heap and array into relation. Fig. 7 illustrates the

```

1 public int classify(int value) {
2     int b = 1;
3     for (int i = 0; i < log2(k); ++i)
4         b = 2 * b + (tree[b] < value ? 1 : 0);
5     return b - k;
6 }

```

Fig. 6: Classifying a single element without branches. The loop at line 3 can be unrolled, because $\log_2 k$ is at most 8. The conditional in line 4 can be compiled into predicated instructions, such as `CMOV`, or, more commonly, into a `CMP/SETcc` sequence, rendering the code effectively branch-free.

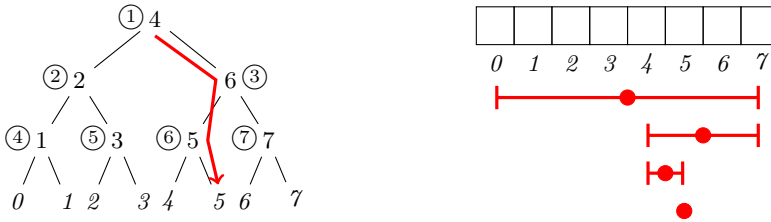


Fig. 7: Visualisation of finding the classification for an element; in the binary heap search tree (left) and in a linearly sorted array (right) for $k = 8$ buckets. The red path indicates the same classification as a path on the heap tree and a nesting of intervals for the binary search. The circled numbers indicate the index in the array representing the search tree; the italic numbers show the bucket number and the upright numbers the index of the splitters against which is compared.

relationship between the search in the binary heap and the search in the ghost code sorted index array.

Besides Classification. The algorithm’s initial step of drawing samples and determining the splitters to be used in the recursion step operates on a fixed number of elements such that most of the properties of this step can be shown by an exhaustive bounded analysis⁵. The permutation and cleanup steps build upon the same general principles already established during classification, but require more and additional book keeping to relate different indices into the array. The implementation consists of four quadruply nested loops and the innermost loop has three different exit paths. Hence, verifying the permutation and cleanup part needed the most proof rule applications to close.

4.3 Selected Cross-cutting Concerns of the Proofs

While constructing the correctness proofs for `ips`⁴o, we made the following noteworthy observations.

⁵ The extended version [3] elaborates on this.

Non-trivial termination proofs. For many algorithms, termination is an easy to show property. However, even though `ips4o` follows essentially an array-based divide-and-conquer strategy, its termination proofs are non-trivial. We exemplify this on the termination of the partitioning step.

The textbook version of quicksort removes the splitter element (pivot) from the partitions. Hence, the partition size is a variant (termination witness) as each recursive call receives a strictly smaller slice to work on. For our `ips4o` implementation, however, this is not the case as the splitter elements remain within the partitions. It is the following observation that ensures termination: If there are two elements e_1, e_2 in the input slice that are classified into two different buckets ($c(e_1) \neq c(e_2)$), then the number of elements in each bucket is strictly below the size of the input slice. While this observation may look trivial to a human reader, it requires a non-trivial interactive proof in KeY. One has to reason that for every bucket b_1 , there is a different non-empty bucket b_2 implying that b_1 is smaller than the input slice. This variant allows proving the termination of the recursion.

Multiple variants of property formalisations. One important insight from the case study is that for some properties it pays off to have not one but two (or multiple) syntactically different, yet semantically equivalent formalisations at hand and to be able to use them at different places in the proofs. We give examples on sortedness and permutation properties.

Sortedness of an array can be expressed in first-order logic by either of the following equivalent formulae:

$$\forall i: 0 \leq i < n - 1 \Rightarrow v[i] \leq v[i + 1] \quad (1)$$

$$\forall i, j: 0 \leq i < n \wedge i \leq j < n \Rightarrow v[i] \leq v[j] \quad (2)$$

While (1) compares every array element with its successor, (2) allows comparison between arbitrary indices in the array. In the case study, when *proving* sortedness, (1) is used. However, when assuming sortedness in a proof (e.g., in preconditions), the transitive representation (2) is more useful. Technically, both representations are formulated as model methods and their equivalence has been shown using a simple inductive argument, which allowed us to switch between representations as needed.

A similar effect with two formalisation variations can be observed for the permutation property: For two sequences s_1, s_2 , the expression `seqPerm`(s_1, s_2) formulates that there exists a bijection π between the indices of s_1 and s_2 such that $s_1[\pi(i)] = s_2[i]$ for all indices i . This straightforward formulation of the property using an explicit permutation witness π proved helpful to show statements like $\sum_{i=0}^n s_1[i] = \sum_{i=0}^n s_2[i]$ under the assumption that s_1 and s_2 are permutations of one another. However, proving the permutation property using this definition can be difficult since one has to provide the explicit witness for π . Therefore, an alternative formulation has been used based on the fact that two sequences are permutations of one another iff they are equal when considered as multisets, i.e., iff every element occurs equally often in both sequences⁶. The equivalence of the two notions is made available to KeY as an (proved) axiom.

⁶ which is a standard formalisation often used in proofs of sorting algorithms

Proving frame conditions. To reason that the memory footprints of different data structures do not overlap, KeY supports the concept of *dynamic frames* [18]. To be cache-efficient, the ips⁴o implementation uses a number of auxiliary buffers, realised as Java arrays. In the Java language, array variables may alias. In the case study, methods have up to 11 array parameters which all must not alias with each other. JML possesses an operator `\disjoint` which can be used to specify that the sets of memory locations provided as arguments must be disjoint. KeY then generates the (quadratically many) inequalities capturing the non-aliasing. KeY is not slowed down since all generated formulas are inequalities between identifiers. We used an auxiliary class to group all arrays for reuse during the recursion which reduced the required specification overhead. This shows that dynamic frames are an adequate formalism to deal with the framing problem for this type of algorithmic verification challenge.

Integer overflow. As mentioned above, KeY uses mathematical integers to model machine `int` values. For this to be sound, arithmetic expressions must not over- or underflow the ranges of their respective primitive type. We hence verified the absence of integer overflows in all methods proved in KeY. Corresponding assertions are automatically generated by KeY during symbolic execution: every arithmetic operation generates a new goal where the absence of overflow for this operation is checked. There were only a few lines of additional specification required. The overwhelming majority of those proofs closed without interactions since they could be derived from already proven invariants.

Performance and Verifiability. Optimisations to the code in the case study sometimes had an impact on the required effort to verify and sometimes did not: verifying the binary search tree optimisation explained in Sec. 4.2 was pretty costly whereas the reverification of the project after the optimisations mentioned in Sec. 3.2 went through pretty automatically. Both optimisations bought a noteworthy bit of performance. A key factor for the complexity of the verification is how much the optimisation modifies data representation.

4.4 Proof Statistics

Table 1 gives an overview of the size of the proofs in this case study. A rule application in the KeY system may be part of the symbolic execution of Java code, part of first-order or theory reasoning.

The overall ratio between specification and source code lines is about 3:1, which since many model methods were declared, is still quite low. Using models methods to formulate lemmas deduplicating the proofs allowed us to obtain an overall proof with only 10^6 steps. Consider in comparison a recent case study [5] performed with KeY: The numbers of branches and rule applications are in the same order of magnitude; but our case study has $6\times$ as many the lines of code, and $7\times$ as many lines of specification. However it also required twice the number of manual interactions.

The specification consists of 179 JML contracts of which 114 could be verified with fewer than ten manual interactions. However, some methods require extensive interaction. Most interactions were needed to prove the contract of a

Table 1: Proof statistics: total number of rule applications, number of interactive rule applications, proof branches, branches closed by calls to an SMT solver, lines of Java code (LOC), lines of JML specification (LOS), ratio LOS/LOC.

Class	Rule apps	Interactions	Branches	SMT	LOC	LOS	$\frac{LOS}{LOC}$
BucketPtrs	206 348	683	585	24	48	441	9.19
Buffers	47 258	120	291	0	44	175	3.98
Classifier	265 743	747	1 540	348	123	481	3.91
Permute	160 431	1 139	1 104	272	130	413	3.18
Cleanup	113 903	485	648	207	102	181	1.77
Sorter	120 079	519	705	7	93	382	4.11
Other	215 629	724	742	44	249	430	1.73
Total	1 015 488	3 932	5 615	789	902	2 503	3.17

Table 2: Most common manual proof interactions in the largest proof (contract of `Permute::swap_block`).

Proof Step	Count	Proof Step	Count
Expanding model method definitions	95	Expanding conditionals	64
Proof state simplification	71	First order equality reasoning	83
Memory footprint reasoning	69	Quantifier instantiation	53
Applying model method contracts	65	Splitting if-then-else expressions	36
		Case distinctions on equalities	35

method wrapping an inner loop from the permutation step with 836 interactions and the cleanup method with 475. Those were also the biggest proofs for method contracts with about 125 000 and 110 000 rule applications, respectively. Without heavy usage of lemma methods, those proofs would have been multiple times larger. Notably, most of the interactions for constructing these proofs were unpacking model methods, using their contracts, simplifying the sequent and using observer dependencies, see Table 2.

5 Performance of the ips⁴o Java Version

As our stated goal is an implementation that is both verified *and* has state-of-the-art efficiency, we performed experiments to measure the performance of our Java implementation of ips⁴o. Our experimental setup is similar to that of the original ips⁴o paper [2] – in particular, we use all of the same input distributions in our evaluation:

- UNIFORM: Values are pseudo-random numbers in $[0, 2^{32}]$.
- ONES: All values are 1.
- SORTED: Values are increasing.
- REVERSED: Values are decreasing.

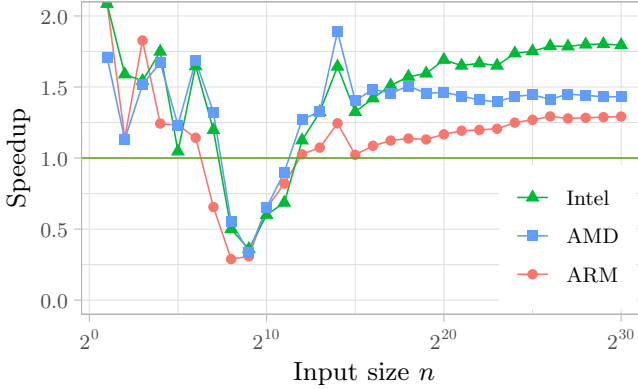


Fig. 8: Speedup of `ips4o` over `Arrays.sort()` for the UNIFORM distribution.

- UNSORTED-TAIL: Like SORTED, except the last $\lfloor \sqrt{n} \rfloor$ elements are shuffled.
- ALMOST-SORTED: Like SORTED, except $\lfloor \sqrt{n} \rfloor$ random adjacent pairs are swapped.
- EXPONENTIAL: Values are distributed exponentially.
- ROOTDUP: Sets $A[i] = i \bmod \lfloor \sqrt{n} \rfloor$.
- TWO DUP: Sets $A[i] = i^2 + \frac{m}{2} \bmod m$, where $m = \lfloor \log_2 n \rfloor$.
- EIGHT DUP: Sets $A[i] = i^8 + \frac{m}{2} \bmod m$, where $m = \lfloor \log_2 n \rfloor$.

We performed experiments using OpenJDK 20 on three different machines/CPUs: An Intel i7 11700 at 4.8 GHz, an AMD Ryzen 3950X at 3.5 GHz, and an Ampere Altra Q80-30 ARM processor at 3 GHz. We repeated each measurement multiple times and report the mean execution times of all iterations. For input sizes $n \leq 2^{13}$, we took 1000 measurements, for $2^{14} \leq n \leq 2^{20}$ we took 25 measurements, and for $2^{21} \leq n \leq 2^{30}$ we took 5 measurements. In addition, we repeated the entire benchmark 5 times to get results across different invocations of the JVM. This means that there are between 25 and 5000 data points for each input size, distribution, and architecture.

On all three machines, `ips4o` outperforms OpenJDK’s `Arrays.sort()` for `int` by a factor of 1.33 to 1.83 for large inputs on the UNIFORM distribution. These results can be found in Fig. 8. For comparison, Fig. 9 shows the runtimes, including the C++ implementation of `ips4o`, on the Intel machine.

Most other distributions show similar results (with a speedup factor of up to 2.27), with the exception of pre-sorted or almost sorted inputs. These distributions – which include ONES, SORTED, REVERSED, and ALMOST-SORTED, but not UNSORTED-TAIL – are detected by the adaptive implementation of `Arrays.sort()` and are not actually sorted by the default dual-pivot quicksort, but by a specialised merging algorithm, which ends up doing almost no work on these distributions.

In summary, our experiments show that the verified Java implementation of `ips4o` outperforms the standard dual-pivot quicksort algorithm across a variety

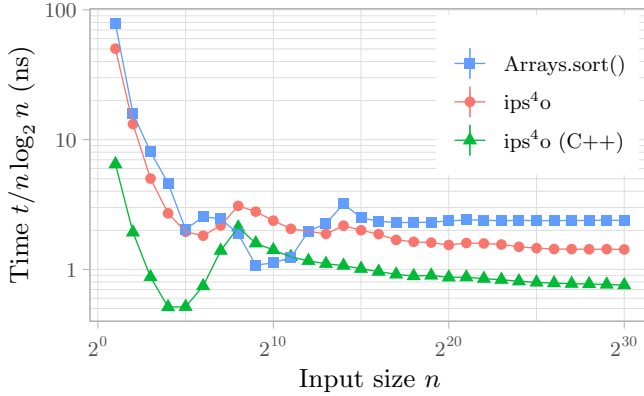


Fig. 9: Runtime for the UNIFORM distribution on Intel.

of input distributions and hardware. The same opportunistic merging algorithm currently implemented by `Arrays.sort()` could be used in conjunction with `ips4o`, which would shortcut the work in case the input is already (almost) sorted.

6 Related Work

JML and KeY have been used previously to verify sorting algorithms. Besides the verifications of nontrivial proof-of-concept implementations like Counting Sort and Radixsort [12], KeY has been used to verify the sorting algorithms deployed with OpenJDK: The formal analysis with KeY revealed a relevant bug in the TimSort implementation shipped with the JDK as the standard algorithm for generic data types [11]. A bugfix was proposed and it was shown that the fixed code does not throw exceptions (but sortedness or permutation were not shown). For the Dual Pivot Quicksort implementation of the JDK (used to sort arrays of primitive values), the sorting and permutation property were successfully specified and verified using KeY [4]. However, the complexity and size of those verification proofs are considerably smaller than our `ips4o` case study. Other pivotal classes of the JDK were also successfully verified using KeY [5,16].

Lammich et al. [20,14] verified efficient sorting routines by proving functional properties on abstract high-level algorithmic descriptions in the Isabelle/HOL theorem prover and then refining them down to LLVM code. In that framework, even parallelised implementations can be analysed to some degree if no shared memory is used [21]. While the verified algorithms are on par with the performance of the standard library, they do not reach the efficiency of `ips4o`, and the authors explicitly list sample sorting as future work. Mohsen and Huisman [34] provide a general framework for the formal verification of swap-based sequential and parallel sorting routines, but restrict it to the analysis of the permutation property. Since `ips4o` is not entirely swap-based (due to the external buffers in the classification step), it is not covered by their approach.

There exists a large number of prominent algorithm verification case studies that focus on the challenges provided by the verification and do not consider the performance of the implementation [8,7,28,17,27,6,26,30].

Finally, there are several large-scale verification projects like the verified microkernel L4.verified [19], the CertiOS framework [37] for the verification of preemptive OS kernels, or the verified Hypervisor Hyper-V [23] that easily top this case study w.r.t. both verified lines of code and invested person years. However, they target a completely different type of system to be verified and have their focus on operating-system-related challenges, like handling concurrent low-level data structures or concurrent accesses to resources. While they also address similar performance questions, the algorithmic aspects are considerably different

7 Conclusions and Future Work

We have demonstrated that a state-of-the-art sorting algorithm like `ips4o` can be formally verified starting directly with an efficient implementation that has not been modified to ease verification. The involved effort of several person months was considerable but seems worthwhile for a widely used basic toolbox function with potential to become part of the standard library of important programming languages. Parts of this verification or at least the basic approach can be reused for related algorithms like radix sort, semisorting, aggregation, hash-join, random permutations, index construction etc.

Future work could look at parallel versions of `ips4o` or implementations that use advanced features such as vector-instructions (e.g., as in [36]). Of course, further basic toolbox components like collection classes (hash tables, search trees etc.) should also be considered.

On the methodology side it would be interesting to compare our approach of direct verification with approaches that start from a verified abstraction of the actual code that is later refined to an implementation. Besides the required effort for verification and the efficiency of the resulting code, a comparison should also consider the ease of communicating with algorithm engineers, which on the one hand may benefit from an abstraction but on the other hand is easier when based on their original implementation. Our case study involved both experts in program verification and experts in algorithm engineering, which proved essential to its success.

For much of the desirable future work, verification tools and methods need further development, in particular for efficient parallel programs and high-performance languages like C++ or Rust. It is also important to better support evolution of the implementation, since it is quite rare that one wants to keep an implementation over decades – algorithm libraries have to evolve with added functionality and changes in hardware, compilers or operating systems.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): *Deductive Software Verification - The KeY Book - From Theory to Prac-*

- tice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). <https://doi.org/10.1007/978-3-319-49812-6>
2. Axtmann, M., Ferizovic, D., Sanders, P., Witt, S.: Engineering in-place (shared-memory) sorting algorithms. *ACM Transaction on Parallel Computing* **9**(1), 2:1–2:62 (2022), see also github.com/ips40. Conference version in ESA 2017
 3. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally verifying an efficient sorter, extended version. Tech. rep., Karlsruhe Institute of Technology (2024). <https://doi.org/10.5445/IR/1000167846>
 4. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK’s dual pivot quick-sort correct. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 35–48. Springer (2017)
 5. Boer, M.d., Gouw, S.d., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK’s identity hash map implementation. In: International Conference on Integrated Formal Methods. pp. 45–62. Springer (2022)
 6. Bottesch, R., Haslbeck, M.W., Thiemann, R.: A verified efficient implementation of the LLL basis reduction algorithm. In: LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16–21 November 2018. pp. 164–180 (2018). <https://doi.org/10.29007/xwww>
 7. Broy, M., Pepper, P.: Combining algebraic and algorithmic reasoning: An approach to the schorr-waite algorithm. *ACM Trans. Program. Lang. Syst.* **4**(3), 362–381 (1982). <https://doi.org/10.1145/357172.357175>
 8. Bubel, R.: The Schorr-Waite-algorithm. In: Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino, pp. 569–587 (2007). https://doi.org/10.1007/978-3-540-69061-0_15
 9. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings. pp. 125–128 (2013). https://doi.org/10.1007/978-3-642-37036-6_8
 10. Frazer, W.D., McKellar, A.C.: Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)* **17**(3), 496–507 (1970)
 11. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK’s sort method for generic collections. *Journal of Automated Reasoning* **62**(1), 93–126 (2019)
 12. de Gouw, S., de Boer, F.S., Rot, J.: Verification of counting sort and radix sort. In: Deductive Software Verification - The KeY Book - From Theory to Practice, pp. 609–618 (2016). https://doi.org/10.1007/978-3-319-49812-6_19
 13. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
 14. Haslbeck, M.P.L., Lammich, P.: For a few dollars more: Verified fine-grained algorithm analysis down to LLVM. *ACM Trans. Program. Lang. Syst.* **44**(3), 14:1–14:36 (2022). <https://doi.org/10.1145/3486169>
 15. Hatchiff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral interface specification languages. *ACM Comput. Surv.* **44**(3), 16:1–16:58 (2012). <https://doi.org/10.1145/2187671.2187678>
 16. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., de Gouw, S.: Verifying OpenJDK’s linkedlist using key (extended paper). *Int. J. Softw. Tools Technol. Transf.* **24**(5), 783–802 (2022). <https://doi.org/10.1007/s10009-022-00679-7>
 17. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7–9 September 2005, Koblenz, Germany. pp. 190–199 (2005). <https://doi.org/10.1109/SEFM.2005.1>

18. Kassios, I.T.: The dynamic frames theory. *Formal Aspects Comput.* **23**(3), 267–288 (2011). <https://doi.org/10.1007/s00165-010-0152-5>
19. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. pp. 207–220 (2009). <https://doi.org/10.1145/1629575.1629596>
20. Lammich, P.: Efficient verified implementation of introsort and pdqsort. In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*. pp. 307–323 (2020). https://doi.org/10.1007/978-3-030-51054-1_18
21. Lammich, P.: Refinement of parallel algorithms down to LLVM. In: *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*. pp. 24:1–24:18 (2022). <https://doi.org/10.4230/LIPIcs.ITP.2022.24>
22. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., et al.: JML reference manual (2008)
23. Leinenbach, D., Santen, T.: Verifying the microsoft Hyper-V hypervisor with VCC. In: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. pp. 806–809 (2009). https://doi.org/10.1007/978-3-642-05089-3_51
24. Leino, K.R.M.: Accessible software verification with Dafny. *IEEE Softw.* **34**(6), 94–97 (2017). <https://doi.org/10.1109/MS.2017.4121212>
25. Leino, K.R.M., Moskal, M.: Usable auto-active verification. *Usable Verification Workshop, Redmond, WS* (2010)
26. Mahboubi, A.: Proving formally the implementation of an efficient gcd algorithm for polynomials. In: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. pp. 438–452 (2006). https://doi.org/10.1007/11814771_37
27. Medina-Bulo, I., Palomo-Lozano, F., Ruiz-Reina, J.: A verified common lisp implementation of Buchberger’s algorithm in ACL2. *J. Symb. Comput.* **45**(1), 96–123 (2010). <https://doi.org/10.1016/j.jsc.2009.07.002>
28. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. In: *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*. pp. 121–135 (2003). https://doi.org/10.1007/978-3-540-45085-6_10
29. Meyer, B.: Applying “design by contract”. *Computer* **25**(10), 40–51 (1992). <https://doi.org/10.1109/2.161279>
30. Mohan, A., Leow, W.X., Hobor, A.: Functional correctness of C implementations of Dijkstra’s, Kruskal’s, and Prim’s algorithms. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. pp. 801–826 (2021). https://doi.org/10.1007/978-3-030-81688-9_37
31. Mommen, N., Jacobs, B.: Verification of C++ programs with VeriFast. *CoRR abs/2212.13754* (2022). <https://doi.org/10.48550/arXiv.2212.13754>
32. Mostowski, W., Ulbrich, M.: Dynamic dispatch for method contracts through abstract predicates. *LNCIS Trans. Modul. Compos.* **1**, 238–267 (2016). https://doi.org/10.1007/978-3-319-46969-0_7
33. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, 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. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings. pp. 257–275 (2020). https://doi.org/10.1007/978-3-030-63461-2_14
 35. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: Autoproof: Auto-active functional verification of object-oriented programs. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings. pp. 566–580 (2015). https://doi.org/10.1007/978-3-662-46681-0_53
 36. Wassenberg, J., Blacher, M., Giesen, J., Sanders, P.: Vectorized and performance-portable quicksort. *Softw. Pract. Exp.* **52**(12), 2684–2699 (2022). <https://doi.org/10.1002/spe.3142>
 37. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part II. pp. 59–79 (2016). https://doi.org/10.1007/978-3-319-41540-6_4

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.





Explainable Online Monitoring of Metric First-Order Temporal Logic

Leonardo Lima^(✉), Jonathan Julián Huerta y Munive, and Dmitriy Traytel^(✉)

Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
{leonardo, traytel}@di.ku.dk

Abstract. Metric first-order temporal logic (MFOTL) is an expressive formalism for specifying temporal and data-dependent constraints on streams of time-stamped, data-carrying events. It serves as the specification language of several runtime monitors. These monitors input an MFOTL formula and an event stream prefix and output satisfying assignments to the formula’s free variables. For complex formulas, it may be unclear why a certain assignment is output. We propose an approach that accompanies assignments with detailed explanations, in the form of proof trees. We develop a new monitor that outputs such explanations. Our tool incorporates a formally verified checker that certifies the explanations and a visualization that allows users to interactively explore and understand the outputs.

1 Introduction

Runtime monitoring is concerned with the analysis of events produced by a system during its execution. An online monitor searches for given complex patterns in event streams, processing the stream incrementally, i.e., one event at a time. If it finds a pattern match, the monitor outputs a verdict to its user. The nature of a verdict depends on both the monitor and its pattern specification language. For propositional specification languages, such as metric temporal logic (MTL) [6, 21], typical verdicts are streams of Booleans [8, 28, 31], where each Boolean signifies the presence or the absence of a pattern match, i.e., the satisfaction or violation of the MTL formula at every position in the input stream.

Users might find Boolean outputs difficult to interpret, especially when complex patterns like nesting temporal operators are involved. In particular, Boolean verdicts give no insight into how monitors produce them—we have to trust their correctness. Even when assuming infallible monitors, verdict justifications can help us to ensure that we expressed correctly our intentions in the specification and, e.g., that it is not vacuously true [23].

Lima et al. [25] propose the use of richer verdicts in an MTL monitor. Specifically, they use proof trees in a dedicated proof system resembling MTL’s semantics to explain why a formula is satisfied or violated. They develop the EXPLANATOR2 monitor, which outputs a stream of size-minimal proof trees, and design an interactive graphical user interface for exploring and understanding these informative verdicts. In addition, they formally verify, in the Isabelle/HOL proof assistant, a proof tree checker certifying that their proof system rules were correctly applied. Thus proof tree verdicts serve a two-fold purpose: as machine-checkable certificates and human-readable explanations.

In this work, we significantly widen the scope of the “proof tree verdicts” approach. We provide certifiable and explainable monitoring verdicts for metric first-order temporal logic (MFOTL) [14] with bounded future operators and without equality between variables. MFOTL extends MTL with data parameters and first-order quantification and is an

expressive formalism with many practical applications [7, 10–13]. We extend Lima et al.’s MTL proof system to MFOTL with the expected rules for quantifiers (Section 2): e.g., the universally quantified formula $\forall x. \alpha$ is satisfied if α with x replaced by d is satisfied *for all* domain values d . The key challenge here is that the domain is typically infinite, which results in the above proof rule for \forall to be infinitely branching. This is problematic because it is unclear how to validate a correct application of the \forall rule in a proof tree checker.

A crucial observation is that without equality between variables, proof trees cannot distinguish values outside of the *active domain*, i.e., the finite set of data values from the monitored event stream prefix and from the formula’s constants. Thus, the active domain’s size plus one bounds the number of choices for d requiring *different* proof trees, and we can reuse them—with the extra “plus one” representing values outside the active domain. Thus, to represent the \forall rule it suffices to store a finite partition of the domain and one subproof for each part. We obtain finite proof objects, develop a checker for them, and formally verify the checker’s correctness in Isabelle/HOL (Section 3).

The proof system explains how to deal with closed MFOTL formulas. A Boolean verdict for a formula with free variables only makes sense relative to a variable assignment. Hence, traditional MFOTL monitors compute sets of satisfying variable assignments [15, 30] instead of Boolean verdicts. In our setting, an explanation for a formula with free variables must provide a proof tree for any variable assignment (satisfying or violating). For infinite domains, there are infinitely many assignments, but the same idea that worked for quantifiers comes to our rescue: it suffices to consider a finite partition of the domain for each variable. Inspired by binary decision diagrams (BDDs) [16], we organize the partitions for different variables hierarchically in *partitioned decision trees* (PDTs). PDTs are trees where each leaf stores a generic data item and each node (representing a variable) branches on a finite partition of the domain (Section 4). The partitions may change from one node to the other. PDTs can be compacted (or reduced in BDD terminology).

We thus have arrived at our notion of *explainable verdicts* for MFOTL formulas: PDTs whose leaves are proof objects. We extend our verified checker from proof objects to such verdicts and Lima et al.’s algorithm for MTL [25] to MFOTL (Section 5). Our algorithm extension is modular in the sense that it merely adds a layer of PDTs, but keeps Lima et al.’s algorithms for temporal operators unchanged. We implement the extended algorithm in a new monitor and also extend Lima et al.’s interactive visualization of proof objects. We demonstrate the effectiveness of our new tool on MFOTL policies from the literature (Section 6). In summary, we make the following contributions:

- We develop a proof system for MFOTL satisfaction and violation at a time-point for a given event stream and verify its soundness and completeness in Isabelle/HOL.
- We finitely represent our proof system’s proof trees and formally verify a checker for them. The key idea is that finite partitions of infinite domains are sufficient.
- We design partitioned decision trees (PDTs) to represent functions from variable assignments to generic data items in a way that enables sharing and compression.
- We develop an algorithm computing explanations: PDTs with proof objects as leaves. We implement the algorithm in a new monitor, along with an interactive visualization of explanations and integrated with the verified proof tree checker for certification.

Our tool, called WHYMON, is publicly available [2].

$v, i \models tt$	$v, i \models \exists x. \alpha$ iff $v[x \mapsto d], i \models \alpha$ for some $d \in \mathbb{D}$
$v, i \not\models ff$	$v, i \models \forall x. \alpha$ iff $v[x \mapsto d], i \models \alpha$ for all $d \in \mathbb{D}$
$v, i \models p(\bar{t})$ iff $p(\llbracket \bar{t} \rrbracket_v) \in \Gamma_i$	$v, i \models \bullet_I \alpha$ iff $i > 0$, $\tau_i - \tau_{i-1} \in I$, and $v, i-1 \models \alpha$
$v, i \models x \approx c$ iff $v(x) = c$	$v, i \models \circ_I \alpha$ iff $\tau_{i+1} - \tau_i \in I$ and $v, i+1 \models \alpha$
$v, i \models \neg \alpha$ iff $v, i \not\models \alpha$	$v, i \models \blacklozenge_I \alpha$ iff $v, j \models \beta$ for some $j \leq i$ with $\tau_i - \tau_j \in I$
$v, i \models \alpha \wedge \beta$ iff $v, i \models \alpha$ and $v, i \models \beta$	$v, i \models \blacklozenge_I \alpha$ iff $v, j \models \beta$ for some $j \geq i$ with $\tau_j - \tau_i \in I$
$v, i \models \alpha \vee \beta$ iff $v, i \models \alpha$ or $v, i \models \beta$	$v, i \models \blacksquare_I \alpha$ iff $v, j \models \beta$ for all $j \leq i$ with $\tau_i - \tau_j \in I$
$v, i \models \alpha \rightarrow \beta$ iff $v, i \not\models \alpha$ or $v, i \models \beta$	$v, i \models \square_I \alpha$ iff $v, j \models \beta$ for all $j \geq i$ with $\tau_j - \tau_i \in I$
$v, i \models \alpha S_I \beta$ iff $v, j \models \beta$ for some $j \leq i$ with $\tau_i - \tau_j \in I$ and $v, k \models \alpha$ for all $j < k \leq i$	
$v, i \models \alpha \mathcal{U}_I \beta$ iff $v, j \models \beta$ for some $j \geq i$ with $\tau_j - \tau_i \in I$ and $v, k \models \alpha$ for all $i \leq k < j$	

Fig. 1: Semantics of MFOTL for a fixed stream $\sigma = \langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$.

Further Related Work Lima et al.’s work [25], which we extend, is based on the work by Basin et al. [9] that employed proof trees as explanations in the context of understanding counterexamples of LTL model checkers. We refer to these works for a discussion of related proof systems for propositional temporal logics and regular expressions.

In the first-order monitoring setting, we are on unexplored territory with verdicts that go beyond satisfying assignments. Nonetheless our work incorporates ideas from existing first-order monitors. Most closely related is Havelund et al.’s DEJAVU monitor [18], which uses BDDs to represent sets of satisfying assignments. Our work generalizes BDDs to branching over partitions of the domain and storing generic data (e.g., proof objects) instead of Booleans in the leaves. In addition, the DEJAVU authors make use of the fact that without equality between variables the formula’s satisfaction cannot be influenced by different values outside the active domain. We generalize this observation so that not only the satisfaction but rather entire proof trees can be reused when exchanging values outside the active domain. Finally, DEJAVU only supports past temporal operators and closed formulas, whereas our algorithm supports both past and future operators and free variables.

Havelund et al.’s key observation fails for equalities between variables. For example, the formula $x \approx y \rightarrow p(x, y)$ is satisfied for any pair of distinct values $c \neq d$ outside of the predicate p ’s interpretation, but it is violated if we pick the same value c for both x and y . A classic result by Ailamazyan et al. [5, 19] shows that for the relational calculus (MFOTL without temporal operators) it suffices to distinguish a finite number of equivalence classes of values outside of the active domain. While it is conceivable that this result generalizes to MFOTL with equality, we leave this generalization as future work.

The MFOTL monitor MonPoly [14, 15] and its formally verified counterpart VeriMon [30] output streams of satisfying assignments for formulas in the so-called monitorable fragment. The fragment ensures that all subformulas always evaluate to finite sets of satisfying assignments. Our monitor does not suffer from this limitation; even more it returns all satisfying and violating assignments (labeled and explained as such).

Outside of first-order monitoring, our visualization takes some inspiration from the stream runtime verification tool TeSSLa [24], which can provide output for all intermediate streams. Similarly, we provide output for all subformulas, but our proof trees allow us to focus on the relevant dependencies between a formula and its subformulas.

Metric first-order temporal logic (MFOTL) We recall MFOTL’s syntax and semantics. We fix an infinite domain \mathbb{D} (e.g., containing integers and strings). Terms $t \in \mathbb{T}$ are either variables $x, y, z \in \mathbb{V}$ or constants $c, d \in \mathbb{D}$. Overlines indicate lists (finite sequences), e.g.,

if t is a term, then \bar{t} is a list of terms. The grammar below specifies MFOTL's syntax, where $p \in \mathbb{E}$ is a predicate name (e.g., a string) and $I \in \mathbb{I} \subseteq 2^{\mathbb{N}}$ is a non-empty interval.

$$\begin{aligned} \alpha ::= & tt \mid \text{ff} \mid p(\bar{t}) \mid x \approx c \mid \neg \alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \alpha \rightarrow \alpha \mid \exists x. \alpha \mid \forall x. \alpha \mid \\ & \bullet_I \alpha \mid \circ_I \alpha \mid \blacklozenge_I \alpha \mid \lozenge_I \alpha \mid \blacksquare_I \alpha \mid \square_I \alpha \mid \alpha \mathcal{S}_I \alpha \mid \alpha \mathcal{U}_I \alpha \end{aligned}$$

Besides the first-order logic operators, the syntax includes the past \bullet (previous), \blacklozenge (once), \blacksquare (historically), \mathcal{S} (since) and future \circ (next), \lozenge (eventually), \square (always), \mathcal{U} (until) temporal operators. We use \wedge for universal and \vee for existential quantification at the metalanguage level to avoid confusion with MFOTL formulas. We also use common interval notation $[a, b) = \{n \mid a \leq n < b\}$ or $[a, c] = \{n \mid a \leq n \leq c\}$, for $a, c \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$, and omit intervals when $I = [0, \infty) = \mathbb{N}$. Whenever we write $[a, c]$, we exclusively denote the range $[a, \dots, c]$ (rather than the two element sequence $[a, c]$). Furthermore, we assume that the future operators (\lozenge , \square , and \mathcal{U}) intervals are finite (also called bounded). We write $a + I$ for $\{a + x \mid x \in I\}$ and $a \mathcal{R} I$ for $\bigwedge x \in I. a \mathcal{R} x$ (where $\mathcal{R} \in \{<, \leq, >, \geq\}$). We interpret formulas over streams σ : infinite sequences of time-stamped sets of events $\sigma = \langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$. We call the indices $i \in \mathbb{N}$ time-points, so that Γ_i is the set of events and $\tau_i \in \mathbb{N}$ is the time-stamp at time-point i . The time-stamps τ_i must be monotone ($\bigwedge i, j. i \leq j \rightarrow \tau_i \leq \tau_j$) and eventually increasing ($\bigwedge \tau. \bigvee i. \tau_i > \tau$). Each event has the form $p(d_1, \dots, d_n)$ where p is the event name and $d_i \in \mathbb{D}$. Given a total assignment v mapping variables to values in \mathbb{D} , we define $\llbracket x \rrbracket_v = v(x)$ and $\llbracket c \rrbracket_v = c$. The notation $\llbracket \bar{t} \rrbracket_v = \bar{c}$ lifts this operation to lists of terms. We define the satisfaction relation $v, i \models_{\sigma} \alpha$ in the usual way (Figure 1). Finally, the *earliest time-point* $\text{ETP}_{\sigma}(\tau)$ of $\tau \in \mathbb{N}$ on σ is the smallest time-point i such that $\tau_i \geq \tau$. Analogously, the *latest time-point* $\text{LTP}_{\sigma}(\tau)$ of $\tau \geq \tau_0$ on σ is the largest i such that $\tau_i \leq \tau$. We omit the stream σ (e.g., \models , $\text{ETP}(\tau)$ and $\text{LTP}(\tau)$) if it is clear from the context.

2 Proof System

We introduce a local proof system for MFOTL (Figure 2). “Local” means here that the proof system does not talk about satisfiability in general, but rather about the formula's satisfaction or violation for a fixed stream, assignment, and time-point.

Our proof system consists of two mutually dependent judgments, \vdash_{σ}^{+} and \vdash_{σ}^{-} (again σ is omitted when clear), that characterize a formula's satisfaction $v, i \vdash_{\sigma}^{+} \alpha$ and violation $v, i \vdash_{\sigma}^{-} \alpha$ relations for assignment v , stream σ , and time-point i . The rules of our proof system closely follow the MFOTL semantics (Figure 1) and extend the proof system used by Lima et al. [25] with assignments (that are mostly passed around without modification) and the rules for quantifiers (which modify the assignments). The rules for atomic predicates and Boolean constants and operators are self-explanatory: e.g., predicates are satisfied if a matching event is present in the trace; a conjunction is satisfied if both conjuncts are satisfied; a conjunction is violated if either of the conjuncts is violated.

The rule \exists^{+} states that for v to satisfy $\exists x. \alpha$ at i , it suffices to provide a domain value d such that the updated assignment $v[x \mapsto d]$ setting x to d satisfies α at i . Conversely, \exists^{-} asserts that the violation of $\exists x. \alpha$ under v at i requires showing that all domain values make $v[x \mapsto d]$ violate α at i . Since the universal quantifier is dual to the existential one, the rules \forall^{-} and \forall^{+} exchange the relations \vdash_{σ}^{+} and \vdash_{σ}^{-} compared to \exists^{+} and \exists^{-} .

$$\begin{array}{c}
\frac{}{v, i \vdash^+ tt} \text{ } tt^+ \quad \frac{}{v, i \vdash^- ff} \text{ } ff^- \quad \frac{\mathbf{p}(\llbracket \bar{t} \rrbracket_v) \in \Gamma_i}{v, i \vdash^+ \mathbf{p}(\bar{t})} p^+ \quad \frac{\mathbf{p}(\llbracket \bar{t} \rrbracket_v) \notin \Gamma_i}{v, i \vdash^- \mathbf{p}(\bar{t})} p^- \quad \frac{v, i \vdash^+ \alpha \quad v, i \vdash^- \beta}{v, i \vdash^- \alpha \rightarrow \beta} \rightarrow^- \\
\frac{v, i \vdash^- \alpha}{v, i \vdash^+ \neg \alpha} \neg^+ \quad \frac{v, i \vdash^- \alpha}{v, i \vdash^- \alpha \wedge \beta} \wedge_L^- \quad \frac{v, i \vdash^- \beta}{v, i \vdash^- \alpha \wedge \beta} \wedge_R^- \quad \frac{v, i \vdash^+ \alpha \quad v, i \vdash^+ \beta}{v, i \vdash^+ \alpha \wedge \beta} \wedge^+ \quad \frac{v, i \vdash^- \alpha}{v, i \vdash^+ \alpha \rightarrow \beta} \rightarrow_L^+ \\
\frac{v, i \vdash^+ \alpha}{v, i \vdash^- \neg \alpha} \neg^- \quad \frac{v, i \vdash^+ \alpha}{v, i \vdash^+ \alpha \vee \beta} \vee_L^+ \quad \frac{v, i \vdash^+ \beta}{v, i \vdash^+ \alpha \vee \beta} \vee_R^+ \quad \frac{v, i \vdash^- \alpha \quad v, i \vdash^- \beta}{v, i \vdash^- \alpha \vee \beta} \vee^- \quad \frac{v, i \vdash^+ \beta}{v, i \vdash^+ \alpha \rightarrow \beta} \rightarrow_R^+ \\
\frac{v[x \mapsto d], i \vdash^+ \alpha}{v, i \vdash^+ \exists x. \alpha} \exists^+ \quad \frac{\wedge d. v[x \mapsto d], i \vdash^+ \alpha}{v, i \vdash^+ \forall x. \alpha} \forall^+ \quad \frac{\wedge d. v[x \mapsto d], i \vdash^- \alpha}{v, i \vdash^- \exists x. \alpha} \exists^- \quad \frac{v[x \mapsto d], i \vdash^- \alpha}{v, i \vdash^- \forall x. \alpha} \forall^- \\
\frac{}{v, 0 \vdash^- \bullet_0} \bullet_0^- \quad \frac{i > 0 \quad \tau_i - \tau_{i-1} < I}{v, i \vdash^- \bullet_{I\alpha}} \bullet_{<I}^- \quad \frac{i > 0 \quad \tau_i - \tau_{i-1} > I}{v, i \vdash^- \bullet_{I\alpha}} \bullet_{>I}^- \quad \frac{i > 0 \quad v, i-1 \vdash^- \alpha}{v, i \vdash^- \bullet_{I\alpha}} \bullet^- \\
\frac{\tau_{i+1} - \tau_i \in I \quad v, i+1 \vdash^+ \alpha}{v, i \vdash^+ \bigcirc_{I\alpha}} \bigcirc^+ \quad \frac{\tau_{i+1} - \tau_i < I}{v, i \vdash^- \bigcirc_{I\alpha}} \bigcirc_{<I}^- \quad \frac{\tau_{i+1} - \tau_i > I}{v, i \vdash^- \bigcirc_{I\alpha}} \bigcirc_{>I}^- \quad \frac{v, i+1 \vdash^- \alpha}{v, i \vdash^- \bigcirc_{I\alpha}} \bigcirc^- \\
\frac{j \leq i \quad \tau_i - \tau_j \in I \quad v, j \vdash^+ \alpha}{v, i \vdash^+ \blacklozenge_{I\alpha}} \blacklozenge^+ \quad \frac{\tau_i < \tau_0 + I}{v, i \vdash^- \blacklozenge_{I\alpha}} \blacklozenge_{<I}^- \quad \frac{j \geq i \quad \tau_j - \tau_i \in I \quad v, j \vdash^+ \alpha}{v, i \vdash^+ \blacklozenge_{I\alpha}} \blacklozenge^+ \\
\frac{\tau_i \geq \tau_0 + I \quad \wedge j \in [\mathbf{E}_i^p(I), \mathbf{L}_i^p(I)]. \quad v, j \vdash^- \alpha}{v, i \vdash^- \blacklozenge_{I\alpha}} \blacklozenge^- \quad \frac{\wedge j \in [\mathbf{E}_i^f(I), \mathbf{L}_i^f(I)]. \quad v, j \vdash^- \beta}{v, i \vdash^- \blacklozenge_{I\alpha}} \blacklozenge^- \\
\frac{j \leq i \quad \tau_i - \tau_j \in I \quad v, j \vdash^- \alpha}{v, i \vdash^- \blacksquare_{I\alpha}} \blacksquare^- \quad \frac{\tau_i < \tau_0 + I}{v, i \vdash^+ \blacksquare_{I\alpha}} \blacksquare_{<I}^+ \quad \frac{j \geq i \quad \tau_j - \tau_i \in I \quad v, j \vdash^- \alpha}{v, i \vdash^+ \square_{I\alpha}} \square^- \\
\frac{\tau_i \geq \tau_0 + I \quad \wedge j \in [\mathbf{E}_i^p(I), \mathbf{L}_i^p(I)]. \quad v, j \vdash^+ \alpha}{v, i \vdash^+ \blacksquare_{I\alpha}} \blacksquare^+ \quad \frac{\wedge j \in [\mathbf{E}_i^f(I), \mathbf{L}_i^f(I)]. \quad v, j \vdash^+ \beta}{v, i \vdash^+ \square_{I\alpha}} \square^+ \\
\frac{j \leq i \quad \tau_i - \tau_j \in I \quad v, j \vdash^+ \beta \quad \wedge k \in (j, i]. \quad v, k \vdash^+ \alpha}{v, i \vdash^+ \alpha S_I \beta} S^+ \quad \frac{i > 0 \quad \tau_i - \tau_{i-1} \in I \quad v, i-1 \vdash^+ \alpha}{v, i \vdash^+ \bullet_{I\alpha}} \bullet^+ \\
\frac{i \leq j \quad \tau_j - \tau_i \in I \quad v, j \vdash^+ \beta \quad \forall k \in [i, j]. \quad v, k \vdash^+ \alpha}{v, i \vdash^+ \alpha \mathcal{U}_I \beta} \mathcal{U}^+ \quad \frac{\tau_i < \tau_0 + I}{v, i \vdash^- \alpha S_I \beta} S_{<I}^- \quad \frac{v(x) = c}{v, i \vdash^+ x \approx c} \approx^+ \\
\frac{\tau_i \geq \tau_0 + I \quad \wedge k \in [\mathbf{E}_i^p(I), \mathbf{L}_i^p(I)]. \quad v, k \vdash^- \beta}{v, i \vdash^- \alpha S_I \beta} S_\infty^- \quad \frac{\wedge k \in [\mathbf{E}_i^f(I), \mathbf{L}_i^f(I)]. \quad v, k \vdash^- \beta}{v, i \vdash^- \alpha \mathcal{U}_I \beta} \mathcal{U}_\infty^- \\
\frac{\mathbf{E}_i^p(I) \leq j \quad j \leq i \quad \tau_i \geq \tau_0 + I \quad v, j \vdash^- \alpha \quad \wedge k \in [j, \mathbf{L}_i^p(I)]. \quad v, k \vdash^- \beta}{v, i \vdash^- \alpha S_I \beta} S^- \\
\frac{i \leq j \quad j < \mathbf{L}_i^f(I) \quad v, j \vdash^- \alpha \quad \forall k \in [\mathbf{E}_i^f(I), j]. \quad v, k \vdash^- \beta}{v, i \vdash^- \alpha \mathcal{U}_I \beta} \mathcal{U}^- \quad \frac{v(x) \neq c}{v, i \vdash^- x \approx c} \approx^-
\end{array}$$

Fig. 2: Local proof system for MFOTL on a fixed stream $\sigma = \langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$.

The rules \blacklozenge^+ and \lozenge^+ are mere restatements of the MFOTL semantics. Since the operators \blacksquare_I and \square_I are respectively dual to \blacklozenge_I and \lozenge_I , their violation rules \blacksquare^- and \square^- once again exchange \vdash_σ^+ and \vdash_σ^- compared to \blacklozenge^+ and \lozenge^+ . The rule $\blacksquare_{<I}^+$ accounts for the vacuous truth of the operator \blacksquare_I near the start of the stream (when no time-points fall within the interval I). Dually, the rule $\blacklozenge_{<I}^-$ asserts the violation of \blacklozenge_I near the start of the stream. The remaining rules \blacklozenge^- , \lozenge^- , \blacksquare^+ , and \square^+ use notation $\mathbf{E}_i^p(I)$, $\mathbf{L}_i^p(I)$, $\mathbf{E}_i^f(I)$, and $\mathbf{L}_i^f(I)$ to refer to time-points of particular interest relative to the current time-point i . Specifically,

for a future formula $\varphi = \mathcal{F}_I \alpha$ with $\mathcal{F} \in \{\circ, \diamond, \square\}$ and interval $I = [a, b]$ or $I = [a, b)$ such that $b \neq \infty$, the formula's semantics at time-point i may need to refer to any time-point with time-stamp in $[\tau_i + a, \dots, \tau_i + b]$. The latest such time-point is $L_i^f(I) = \text{LTP}(\tau_i + b)$ while the earliest one is $E_i^f(I) = \max(i, \text{ETP}(\tau_i + a))$. For past operators $\mathcal{P} \in \{\bullet, \blacklozenge, \blacksquare\}$, the relevant time-stamp interval is $[\tau_i - b, \dots, \tau_i - a]$ and the interval's earliest time-point is $E_i^p(I) = \text{ETP}(\tau_i - b)$ and its latest time-point is $L_i^p(I) = \min(i, \text{LTP}(\tau_i - a))$.

Proof trees emerging from repeated application of the rules in our proof system contain all the necessary information to explain why a formula is satisfied or violated. In other words, our proof system is sound and complete, i.e., the following result holds.

Theorem 1. *Let α be a formula, v a variable assignment, $i \in \mathbb{N}$ a time-point, and $\sigma = \langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$ a trace. Then $v, i \vdash_{\sigma}^+ \alpha \longleftrightarrow v, i \models_{\sigma} \alpha$ and $v, i \vdash_{\sigma}^- \alpha \longleftrightarrow v, i \not\models_{\sigma} \alpha$.*

We have formalized and verified this result in Isabelle/HOL.

Example 1. Consider the standard publish–approve example [14] requiring that any file f published by an author a , must first be approved by a manager m of a within the previous seven days. The formalization of this policy as a closed MFOTL formula is:

$$\varphi = \forall a. \forall f. \text{publish}(a, f) \rightarrow (\blacklozenge_{[0,7]} \exists m. (\neg \text{mgr}_F(m, a) \mathcal{S} \text{mgr}_S(m, a)) \wedge \text{approve}(m, f)).$$

Here, the events $\text{mgr}_S(m, a)$ and $\text{mgr}_F(m, a)$ mark m starting and finishing being a 's manager. Formally, m is currently a manager of a if m started being a 's manager in the past and has not finished being a 's manager since. Thus, the manager relation changes over time. Consider the stream $\langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$, where $\tau_0 = \tau_1 = 0$, $\tau_2 = 4$, $\tau_3 = 10$, and

$$\begin{aligned} \Gamma_0 &= \{\text{mgr}_S(\text{Mallory}, \text{Alice}), \text{mgr}_S(\text{Merlin}, \text{Bob}), \text{mgr}_S(\text{Merlin}, \text{Charlie})\}, \text{ and} \\ \Gamma_1 &= \{\text{approve}(\text{Mallory}, 152)\}, \text{ and} \\ \Gamma_2 &= \{\text{approve}(\text{Merlin}, 163), \text{publish}(\text{Alice}, 160), \text{mgr}_F(\text{Merlin}, \text{Charlie})\}, \text{ and} \\ \Gamma_3 &= \{\text{approve}(\text{Merlin}, 187), \text{publish}(\text{Bob}, 163), \text{publish}(\text{Alice}, 163), \\ &\quad \text{publish}(\text{Charlie}, 163), \text{publish}(\text{Charlie}, 152)\}. \end{aligned}$$

In the following we abbreviate the subformulas of φ as follows: $\varphi_L = \text{publish}(a, f)$, $\varphi_1 = \neg \text{mgr}_F(m, a) \mathcal{S} \text{mgr}_S(m, a)$, $\varphi_2 = \text{approve}(m, f)$, $\varphi_3 = \exists m. \varphi_1 \wedge \varphi_2$, $\varphi_R = \blacklozenge_{[0,7]} \varphi_3$, and $\varphi' = \varphi_L \rightarrow \varphi_R$. The following proof tree shows that φ is violated at time-point 3 for any v :

$$\begin{array}{c} \text{approve}(d, 152) \notin \Gamma_i \\ \hline v[a \mapsto \text{Charlie}, f \mapsto 152, m \mapsto d], i \vdash^- \varphi_2 \quad P^- \\ \hline v[a \mapsto \text{Charlie}, f \mapsto 152, m \mapsto d], i \vdash^- \varphi_1 \wedge \varphi_2 \quad \wedge_R^- \\ \hline \text{publish}(\text{Charlie}, 152) \in \Gamma_3 \quad P^+ \quad v[a \mapsto \text{Charlie}, f \mapsto 152], i \vdash^- \varphi_3 \quad \blacklozenge^- \\ \hline v[a \mapsto \text{Charlie}, f \mapsto 152], 3 \vdash^+ \varphi_L \quad v[a \mapsto \text{Charlie}, f \mapsto 152], 3 \vdash^- \varphi_R \quad \rightarrow^- \\ \hline v[a \mapsto \text{Charlie}, f \mapsto 152], 3 \vdash^- \varphi_L \rightarrow \varphi_R \quad \rightarrow^- \\ \hline v[a \mapsto \text{Charlie}], 3 \vdash^- \forall f. \varphi' \quad \forall^- \\ \hline v, 3 \vdash^- \forall a. \forall f. \varphi' \quad \forall^- \end{array}$$

Given φ_R 's temporal constraint, we note that $\tau_3 \geq 0$ and need to check $v, i \vdash^- \varphi_3$ for the time-points $i \in \{2, 3\}$ (as $[E_3^p([0, 7]), L_3^p([0, 7])] = \{2, 3\}$). Both subproofs are identical, so we parameterize them over i . In addition, the \exists^- subproofs are valid for an arbitrary manager $d \in \mathbb{D}$ (abbreviating infinite branching over all possible domain values). \square

$$\begin{aligned}
\text{sp} = & \text{tt}^+(\mathbb{N}) \mid p^+(\mathbb{N}, \mathbb{E}, \bar{i}) \mid \neg^+(\text{vp}) \mid \wedge^+(\text{sp}, \text{sp}) \mid \vee_L^+(\text{sp}) \mid \vee_R^+(\text{sp}) \mid \rightarrow^+(\text{vp}) \mid \rightarrow_R^+(\text{sp}) \\
& \mid \forall^+(\mathbb{V}, \mathbb{D}, \mathbb{D}, \text{sp}) \mid \exists^+(\mathbb{V}, \mathbb{D}, \text{sp}) \mid \bullet^+(\text{sp}) \mid \circ^+(\text{sp}) \mid \blacklozenge^+(\mathbb{N}, \text{sp}) \mid \blacklozenge^+(\mathbb{N}, \text{sp}) \\
& \mid \blacksquare_{\leq_I}^+(\mathbb{N}) \mid \blacksquare^+(\mathbb{N}, \overline{\text{sp}}) \mid \square^+(\mathbb{N}, \overline{\text{sp}}) \mid \mathcal{S}^+(\text{sp}, \overline{\text{sp}}) \mid \mathcal{U}^+(\text{sp}, \overline{\text{sp}}), \\
\text{vp} = & \text{ff}^-(\mathbb{N}) \mid p^-(\mathbb{N}, \mathbb{E}, \bar{i}) \mid \neg^-(\text{sp}) \mid \wedge_R^-(\text{vp}) \mid \wedge_L^-(\text{vp}) \mid \vee^-(\text{vp}, \text{vp}) \mid \rightarrow^-(\text{sp}, \text{vp}) \\
& \mid \forall^-(\mathbb{V}, \mathbb{D}, \text{vp}) \mid \exists^-(\mathbb{V}, \mathbb{D}, \text{vp}) \mid \bullet^-(\text{vp}) \mid \bullet_{\leq_I}^-(\mathbb{N}) \mid \bullet_{\geq_I}^-(\mathbb{N}) \mid \bullet_{\bar{o}}^-(\text{vp}) \mid \circ^-(\text{vp}) \mid \circ_{\leq_I}^-(\mathbb{N}) \\
& \mid \circ_{\geq_I}^-(\mathbb{N}) \mid \blacklozenge^-(\mathbb{N}, \overline{\text{vp}}) \mid \blacklozenge_{\leq_I}^-(\mathbb{N}) \mid \blacklozenge_{\geq_I}^-(\mathbb{N}, \overline{\text{vp}}) \mid \blacksquare^-(\mathbb{N}, \text{vp}) \mid \square^-(\mathbb{N}, \text{vp}) \\
& \mid \mathcal{S}_{\leq_I}^-(\mathbb{N}) \mid \mathcal{S}^-(\mathbb{N}, \text{vp}, \overline{\text{vp}}) \mid \mathcal{S}_{\infty}^-(\mathbb{N}, \overline{\text{vp}}) \mid \mathcal{U}^-(\mathbb{N}, \text{vp}, \overline{\text{vp}}) \mid \mathcal{U}_{\infty}^-(\mathbb{N}, \overline{\text{vp}})
\end{aligned}$$

Fig. 3: Grammar for our proof objects.

3 Proof Object Checker

This section introduces our proof objects and their checker: finite data-representations of our proof system’s trees, and an algorithm that certifies if a given proof object faithfully proves the satisfaction or violation of a formula under a given assignment and stream. We discuss the soundness, completeness, and executability of these constructions.

To algorithmically manipulate proof trees, we define an explicit representation of *satisfactions* sp and *violations* vp via the grammar in Figure 3, where each constructor corresponds to a proof rule of our proof system (Figure 2), and its arguments represent subproofs and parameters that are part of a rule. The disjoint union $\mathfrak{p} = \text{sp} \uplus \text{vp}$ is our type of *proof objects*. The proof object \forall^+ requires information about satisfactions for all domain elements $d \in \mathbb{D}$ which we finitely represent with our *valued partitions* $P \in \mathbb{D}(\text{sp})$. Recall that a partition P of a set A is a collection of non-empty, pair-wise disjoint subsets of A that cover A . That is, $D_i \cap D_j = \emptyset$ for $D_i, D_j \in P$ with $D_i \neq D_j$ and $\bigcup P = A$. Partitions enable us to finitely represent all elements of the domain using finitely many finite sets and the co-finite complement of their union. In valued partitions $P \in \mathbb{D}(\text{sp})$, each set in the partition is tagged with a satisfaction explaining why its elements satisfy the argument of a universally quantified formula. Formally, our valued partitions $P \in \mathbb{D}(\text{sp})$ are lists of pairs of a set D_i and a value $z \in Z$ from a given set Z such that the sets D_i form a partition of \mathbb{D} . Similarly, \exists^- stores a valued partition $P \in \mathbb{D}(\text{vp})$ of violations.

Our proof objects $p \in \mathfrak{p}$ represent satisfactions or violations at a certain time-point. We define a function $\text{tp}(p)$ (omitted) to compute this time-point. Either this information can be obtained recursively (e.g., $\text{tp}(\circ^+(p)) = \text{tp}(p) - 1$) or, in cases where it cannot, it is stored directly in the proof objects (e.g., $\text{tp}(\text{tt}^+(i)) = i$). We lift tp to sequences (yielding sequences of time-points) and valued partitions as $\text{tp}(P) = \text{tp}(p_1)$, where (D_1, p_1) is the partition P ’s first entry. To characterize *valid* proof objects, we define the relation \vdash_σ (Figure 4) that checks that proof objects constitute correct applications of our proof system’s rules. Here, \vdash is not an executable algorithm yet since the proof objects \forall^+ and \exists^- require a recursive call for each element of each set in the partition, and at least one of such sets is infinite for infinite domains. We will improve on this aspect after an example. *Example 2.* The following violation proof object p at time-point 3 (i.e., $\text{tp}(p) = 3$) is valid for formula φ on stream σ from Example 1 (i.e., $v, p \vdash_\sigma \varphi$ for any assignment v):

$$\begin{aligned}
p &= \forall^-(a, \text{Charlie}, \forall^-(f, 152, p_{\bar{o}}^-)), \text{ where} \\
p_{\bar{o}}^- &= \rightarrow^-(p_L^+, p_{\blacklozenge}^-), p_L^+ = p^+(3, \text{publish}, [a, f]), \\
p_{\blacklozenge}^- &= \blacklozenge^-(3, [\exists^-(x, [(\mathbb{D}, p_2^-)]), \exists^-(x, [(\mathbb{D}, p_3^-)])]), \text{ and} \\
p_i^- &= \wedge_R^-(p^-(i, \text{approve}, [m, f])) \text{ for } i \in \{2, 3\}.
\end{aligned}$$

$v, tt^+(i) \vdash tt$		$v, ff^-(i) \vdash ff$
$v, p^+(i, p, \bar{t}) \vdash p(\bar{t})$	iff $p(\llbracket \bar{t} \rrbracket_v) \in \Gamma_i$	$v, p^-(i, p, \bar{t}) \vdash p(\bar{t})$ iff $p(\llbracket \bar{t} \rrbracket_v) \notin \Gamma_i$
$v, \approx^+(i, x, c) \vdash x \approx c$	iff $v(x) = c$	$v, \approx^-(i, x, c) \vdash x \approx c$ iff $v(x) \neq c$
$v, \neg^+(vp) \vdash \neg \alpha$	iff $v, vp \vdash \alpha$	$v, \neg^-(sp) \vdash \neg \alpha$ iff $v, sp \vdash \alpha$
$v, \rightarrow_L^+(vp) \vdash \alpha \rightarrow \beta$	iff $v, vp \vdash \alpha$	$v, \wedge_L^-(vp) \vdash \alpha \wedge \beta$ iff $v, vp \vdash \alpha$
$v, \rightarrow_R^+(sp) \vdash \alpha \rightarrow \beta$	iff $v, sp \vdash \beta$	$v, \wedge_R^-(vp) \vdash \alpha \wedge \beta$ iff $v, vp \vdash \beta$
$v, \exists^+(x, d, sp) \vdash \exists x. \alpha$	iff $v[x \mapsto d], sp \vdash \alpha$	$v, \vee_L^-(sp) \vdash \alpha \vee \beta$ iff $v, sp \vdash \alpha$
$v, \forall^-(x, d, vp) \vdash \forall x. \alpha$	iff $v[x \mapsto d], vp \vdash \alpha$	$v, \vee_R^-(sp) \vdash \alpha \vee \beta$ iff $v, sp \vdash \beta$
$v, \wedge^+(sp_1, sp_2) \vdash \alpha \wedge \beta$	iff $v, sp_1 \vdash \alpha$ and $v, sp_2 \vdash \beta$ and $\text{tp}(sp_1) = \text{tp}(sp_2)$	
$v, \vee^-(vp_1, vp_2) \vdash \alpha \vee \beta$	iff $v, vp_1 \vdash \alpha$ and $v, vp_2 \vdash \beta$ and $\text{tp}(vp_1) = \text{tp}(vp_2)$	
$v, \rightarrow^-(sp_1, vp_2) \vdash \alpha \rightarrow \beta$	iff $v, sp_1 \vdash \alpha$ and $v, vp_2 \vdash \beta$ and $\text{tp}(sp_1) = \text{tp}(vp_2)$	
$v, \forall^+(x, P) \vdash \forall x. \alpha$	iff $\bigwedge (D_k, sp_k) \in P. \text{tp}(sp_k) = \text{tp}(P)$ and $\bigwedge d \in D_k. v[x \mapsto d], sp_k \vdash \alpha$	
$v, \exists^-(x, P) \vdash \exists x. \alpha$	iff $\bigwedge (D_k, vp_k) \in P. \text{tp}(vp_k) = \text{tp}(P)$ and $\bigwedge d \in D_k. v[x \mapsto d], vp_k \vdash \alpha$	
$v, \bullet^+(sp) \vdash \bullet_I \alpha$	iff $v, sp \vdash \alpha$ and $\text{tp}(\bullet^+(sp)) = \text{tp}(sp) + 1$ and $\tau_{\text{tp}(\bullet^+(sp))} - \tau_{\text{tp}(sp)} \in I$	
$v, \circ^+(sp) \vdash \circ_I \alpha$	iff $v, sp \vdash \alpha$ and $\text{tp}(\circ^+(sp)) + 1 = \text{tp}(sp)$ and $\tau_{\text{tp}(sp)} - \tau_{\text{tp}(\circ^+(sp))} \in I$	
$v, \blacklozenge^+(i, sp) \vdash \blacklozenge_I \alpha$	iff $v, sp \vdash \alpha$ and $i \geq \text{tp}(sp)$ and $\tau_i - \tau_{\text{tp}(sp)} \in I$	
$v, \blacklozenge^-(i, sp) \vdash \blacklozenge_I \alpha$	iff $v, sp \vdash \alpha$ and $i \leq \text{tp}(sp)$ and $\tau_{\text{tp}(sp)} - \tau_i \in I$	
$v, \blacksquare^+(i, \overline{sp}) \vdash \blacksquare_I \alpha$	iff $(\bigwedge sp \in \overline{sp}. v, sp \vdash \alpha)$ and $\text{tp}(\overline{sp}) = [\text{E}_i^p(I), \text{L}_i^p(I)]$ and $\tau_i \geq \tau_0 + I$	
$v, \square^+(i, \overline{sp}) \vdash \square_I \alpha$	iff $(\bigwedge sp \in \overline{sp}. v, sp \vdash \alpha)$ and $\text{tp}(\overline{sp}) = [\text{E}_i^f(I), \text{L}_i^f(I)]$	
$v, \mathcal{S}^+(sp, \overline{sp}) \vdash \alpha \mathcal{S}_I \beta$	iff $(\bigwedge sp' \in \overline{sp}. v, sp' \vdash \alpha)$ and $v, sp \vdash \beta$ and $\text{tp}(\mathcal{S}^+(sp, \overline{sp})) \geq \text{tp}(sp)$ and $\text{tp}(\overline{sp}) = [\text{tp}(sp) + 1, \text{tp}(\mathcal{S}^+(sp, \overline{sp}))]$ and $\tau_{\text{tp}(\mathcal{S}^+(sp, \overline{sp}))} - \tau_{\text{tp}(sp)} \in I$	
$v, \mathcal{U}^+(sp, \overline{sp}) \vdash \alpha \mathcal{U}_I \beta$	iff $(\bigwedge sp' \in \overline{sp}. v, sp' \vdash \alpha)$ and $v, sp \vdash \beta$ and $\text{tp}(\mathcal{U}^+(sp, \overline{sp})) \leq \text{tp}(sp)$ and $\text{tp}(\overline{sp}) = [\text{tp}(\mathcal{U}^+(sp, \overline{sp})), \text{tp}(sp)]$ and $\tau_{\text{tp}(sp)} - \tau_{\text{tp}(\mathcal{U}^+(sp, \overline{sp}))} \in I$	
$v, \bullet_0^- \vdash \bullet_I \alpha$	iff $\text{tp}(\bullet_0^-) = 0$	$v, \bullet_{<I}^-(i) \vdash \bullet_I \alpha$ iff $i > 0$ and $\tau_i - \tau_{i-1} < I$
$v, \circ_{<I}^-(i) \vdash \circ_I \alpha$	iff $\tau_{i+1} - \tau_i < I$	$v, \bullet_{>I}^-(i) \vdash \bullet_I \alpha$ iff $i > 0$ and $\tau_i - \tau_{i-1} > I$
$v, \circ_{>I}^-(i) \vdash \circ_I \alpha$	iff $\tau_{i+1} - \tau_i > I$	$v, \blacksquare_{<I}^-(i) \vdash \blacksquare_I \alpha$ iff $\tau_i < \tau_0 + I$
$v, \blacklozenge_{<I}^-(i) \vdash \blacklozenge_I \alpha$	iff $\tau_i < \tau_0 + I$	$v, \mathcal{S}_{<I}^-(i) \vdash \alpha \mathcal{S}_I \beta$ iff $\tau_i < \tau_0 + I$
$v, \bullet_{<I}^-(vp) \vdash \bullet_I \alpha$	iff $v, vp \vdash \alpha$ and $\text{tp}(\bullet_{<I}^-(vp)) = \text{tp}(vp) + 1$	
$v, \circ_{<I}^-(vp) \vdash \circ_I \alpha$	iff $v, vp \vdash \alpha$ and $\text{tp}(\circ_{<I}^-(vp)) + 1 = \text{tp}(vp)$	
$v, \blacklozenge_{<I}^-(i, \overline{vp}) \vdash \blacklozenge_I \alpha$	iff $(\bigwedge vp \in \overline{vp}. v, vp \vdash \alpha)$ and $\text{tp}(\overline{vp}) = [\text{E}_i^p(I), \text{L}_i^p(I)]$ and $\tau_i \geq \tau_0 + I$	
$v, \blacklozenge_{>I}^-(i, \overline{vp}) \vdash \blacklozenge_I \alpha$	iff $(\bigwedge vp \in \overline{vp}. v, vp \vdash \alpha)$ and $\text{tp}(\overline{vp}) = [\text{E}_i^f(I), \text{L}_i^f(I)]$	
$v, \blacksquare_{<I}^-(i, \overline{vp}) \vdash \blacksquare_I \alpha$	iff $v, vp \vdash \alpha$ and $i \geq \text{tp}(vp)$ and $\tau_i - \tau_{\text{tp}(vp)} \in I$	
$v, \square_{<I}^-(i, \overline{vp}) \vdash \square_I \alpha$	iff $v, vp \vdash \alpha$ and $i \leq \text{tp}(vp)$ and $\tau_{\text{tp}(vp)} - \tau_i \in I$	
$v, \mathcal{S}_{<I}^-(i, \overline{vp}) \vdash \alpha \mathcal{S}_I \beta$	iff $(\bigwedge vp \in \overline{vp}. v, vp \vdash \beta)$ and $\text{tp}(\overline{vp}) = [\text{E}_i^p(I), \text{L}_i^p(I)]$ and $\tau_i \geq \tau_0 + I$	
$v, \mathcal{S}_{>I}^-(i, \overline{vp}) \vdash \alpha \mathcal{S}_I \beta$	iff $(\bigwedge vp \in \overline{vp}. v, vp \vdash \beta)$ and $v, vp \vdash \alpha$ and $\text{E}_i^p(I) \leq \text{tp}(vp) \leq i$ and $\text{tp}(\overline{vp}) = [\text{tp}(vp), \text{L}_i^p(I)]$ and $\tau_i \geq \tau_0 + I$	
$v, \mathcal{U}_{<I}^-(i, \overline{vp}) \vdash \alpha \mathcal{U}_I \beta$	iff $(\bigwedge vp \in \overline{vp}. v, vp \vdash \beta)$ and $\text{tp}(\overline{vp}) = [\text{E}_i^f(I), \text{L}_i^f(I)]$	
$v, \mathcal{U}_{>I}^-(i, \overline{vp}) \vdash \alpha \mathcal{U}_I \beta$	iff $(\bigwedge vp \in \overline{vp}. v, vp \vdash \beta)$ and $v, vp \vdash \alpha$ and $i \leq \text{tp}(vp) < \text{L}_i^f(I)$ and $\text{tp}(\overline{vp}) = [\text{E}_i^f(I), \text{tp}(vp)]$	

Fig. 4: Proof checker for a fixed stream $\sigma = \langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$.

Indeed, we use the definition in Figure 4 to certify that $v, p \vdash_\sigma \varphi$:

$$\begin{aligned}
v, p \vdash \varphi &\text{ iff } v[a \mapsto \text{Charlie}], \forall^-(f, 152, p_-) \vdash \forall f. \varphi_L \rightarrow \varphi_R \\
&\text{ iff } v[a \mapsto \text{Charlie}, f \mapsto 152], p_- \vdash \varphi_L \rightarrow \varphi_R \\
&\text{ iff } v[a \mapsto \text{Charlie}, f \mapsto 152], p_L^+ \vdash \varphi_L \text{ and } \text{tp}(p_L^+) = 3 = \text{tp}(p_-) \text{ and} \\
&\quad v[a \mapsto \text{Charlie}, f \mapsto 152], p_- \vdash \varphi_R \\
&\text{ iff } v[a \mapsto \text{Charlie}, f \mapsto 152], \exists^-(x, [\mathbb{D}, p_i^-]) \vdash \varphi_{\exists} \text{ for } i \in \{2, 3\} \\
&\text{ iff } v[a \mapsto \text{Charlie}, f \mapsto 152, x \mapsto d], p_i^- \vdash \varphi_1 \wedge \varphi_2 \text{ for all } d \in \mathbb{D}, i \in \{2, 3\} \\
&\text{ iff } \text{approve}(d, 152) \notin \Gamma_i \text{ for all } d \in \mathbb{D}, i \in \{2, 3\}, \text{ which is true.}
\end{aligned}$$

$$\begin{aligned}
\text{LRTP } i \text{ tt} &= \text{LRTP } i \text{ ff} = \text{LRTP } i (\text{p}(\bar{t})) = \text{LRTP } i (x \approx c) = i, \\
\text{LRTP } i (Qx. \alpha) &= \text{LRTP } i (\neg \alpha) = \text{LRTP } i \alpha \text{ for } Q \in \{\forall, \exists\}, \\
\text{LRTP } i (\alpha \oplus \beta) &= \max (\text{LRTP } i \alpha) (\text{LRTP } i \beta) \text{ for } \oplus \in \{\vee, \wedge, \rightarrow\}, \\
\text{LRTP } i (\bullet_I \alpha) &= \text{LRTP } (i-1) \alpha, \quad \text{LRTP } i (\circ_I \alpha) = \text{LRTP } (i+1) \alpha, \\
\text{LRTP } i (\diamond_I \alpha) &= \text{LRTP } i (\square_I \alpha) = \text{LRTP } (\text{L}_i^f(I)) \alpha, \\
\text{LRTP } i (\blacklozenge_I \alpha) &= \text{LRTP } i (\blacksquare_I \alpha) = \text{LRTP } (\text{LTP}_i^{\text{past}} I) \alpha, \\
\text{LRTP } i (\alpha \mathcal{S}_I \beta) &= \max (\text{LRTP } i \alpha) (\text{LRTP } (\text{LTP}_i^{\text{past}} I) \beta), \\
\text{LRTP } i (\alpha \mathcal{U}_I \beta) &= \max (\text{LRTP } (\text{L}_i^f(I) - 1) \alpha) (\text{LRTP } (\text{L}_i^f(I)) \beta), \text{ where} \\
\text{LTP}_i^{\text{past}} I &= (\text{if } \tau_i \geq \tau_0 + I \text{ then } \text{L}_i^p(I) \text{ else } 0).
\end{aligned}$$

Fig. 5: The formula's latest relevant time-point at i for a fixed stream $\sigma = \langle \tau_i, \Gamma_i \rangle_{i \in \mathbb{N}}$.

We implicitly use in the above the true statements $\text{publish}(\text{Charlie}, 152) \in \Gamma_3$, $0 \leq \tau_3$, and $\text{tp}([\exists^-(x, [\mathbb{D}, p_2^-])], \exists^-(x, [\mathbb{D}, p_3^-])) = [2, 3] = [\text{E}_i^p(I), \text{L}_i^p(I)]$. \square

Theorem 2. *Fix a stream σ . The relation \vdash is sound and complete in the sense that $v, i \models \alpha$ iff there is a satisfaction sp such that $v, sp \vdash \alpha$ and $\text{tp}(sp) = i$. Similarly $v, i \not\models \alpha$ iff there is a violation vp such that $v, vp \vdash \alpha$ and $\text{tp}(vp) = i$.*

We have established the above result in Isabelle. Below we sketch our overall approach and highlight the main challenge. We show both soundness and completeness by relating proof object validity (\vdash) to the proof system (\vdash^+ and \vdash^-), which we already know to be sound and complete, i.e., related to the semantics \models . Soundness is easy as the proof object directly provides the recipe for correctly applying the proof system rules. Formally, if $v, sp \vdash \alpha$ then $v, \text{tp}(sp) \vdash^+ \alpha$, and if $v, vp \vdash \alpha$, then $v, \text{tp}(vp) \vdash^- \alpha$. The proof follows immediately by mutual induction on the proof object structure.

Completeness of \vdash requires us to provide a valid proof object just from knowing $v, i \vdash^+ \alpha$ or $v, i \vdash^- \alpha$. We proceed by mutual induction on the derivations of \vdash^+ and \vdash^- . Only two of the quantifier cases are challenging. For the satisfaction of the universal quantifier (and similarly for the violation of \exists), we must construct a valued partition with finitely many subproofs. However, the induction hypothesis yields a separate proof object for every element of the domain \mathbb{D} , and all these proof objects may a priori be different. The crucial observation is that for all values that do not occur in the stream (or at least are not in reach of α with respect to a time-point i) we can reuse the same proof object. To formalize this observation, we first define a formula's *active domain* at i , written $\text{AD}_i(\alpha)$, which formalizes the in “reach” intuition. To this end, we first define the *latest relevant time point* ($\text{LRTP } i \alpha$) of α at i (Figure 5). Intuitively, $\text{LRTP } i \alpha$ marks the largest time-point that may influence α 's satisfiability at i . It exists, because we assume that future temporal operators have bounded intervals. Based on this, we define:

$$\text{AD}_i(\alpha) = \mathbb{D}(\alpha) \cup \bigcup_{k \leq \text{LRTP } i \alpha} \{d \mid d \text{ appears in some } \text{p}(d_1, \dots, d_n) \in \Gamma_k\}.$$

Here we write $\mathbb{D}(\alpha)$ for the set of constants $d \in \mathbb{D}$ occurring in subformulas of the form $x \approx d$ in α . (In contrast to constants occurring in atomic predicates, constants occurring in equalities may appear in α 's satisfying assignments even if they are not part of the trace.) Note that $\text{AD}_i(\alpha)$ is finite. The active domain lets us formalize the key observation:

Lemma 1. *Fix a stream σ , a formula α , a proof p , and two assignments v and v' . Let $i = \text{tp}(p)$, $\text{AD} = \text{AD}_i(\alpha)$, and V be the set of α 's free variables. Assume that v and v'*

may only disagree on V for values outside of the active domain at i , i.e.,

$$\forall x \in V. v(x) = v'(x) \vee (v(x) \notin \text{AD} \wedge v'(x) \notin \text{AD}).$$

Then, p 's validity status is the same for both assignments, i.e., $v, p \vdash \alpha$ iff $v', p \vdash \alpha$.

We now can finish the \forall^+ case of the completeness proof. By the induction hypothesis, there is a satisfaction $p(d) \in \mathfrak{sp}$ for each domain element $d \in \mathbb{D}$. Moreover, $\{\{d\} \mid d \in \text{AD}_i(\alpha)\} \cup \{\mathbb{D} \setminus \text{AD}_i(\alpha)\}$ is a finite partition of \mathbb{D} . Hence, the list of pairings $(\{d\}, p(d))$ for each $d \in \text{AD}_i(\alpha)$ and $(\mathbb{D} \setminus \text{AD}_i(\alpha), p(z))$ for some $z \in \mathbb{D} \setminus \text{AD}_i(\alpha)$ (which exists as \mathbb{D} is infinite) is a valued partition. Moreover, all subproofs are valid for all the values contained in the partition sets by combining the induction hypothesis with the above congruence Lemma 1 (for $p = p(z)$), and thus so is the overall \forall^+ proof object.

Lastly, we address the executability issue. The validity relation \vdash works with assignments v of values to variables. To avoid performing infinitely many recursive calls for the \forall^+ and \exists^- proof objects we now will work with *set assignments* V of sets of values to variables. We define a validity relation $V, p \vdash \alpha$ based on set assignments. The definition is the same as the one of $v, p \vdash \alpha$ except for the predicate and the quantifier cases:

$$\begin{aligned} V, p^+(i, p, \bar{t}) \vdash p(\bar{t}) & \quad \text{iff } \{p\} \times \llbracket \bar{t} \rrbracket_V \subseteq \Gamma_i \\ V, p^-(i, p, \bar{t}) \vdash p(\bar{t}) & \quad \text{iff } \{p\} \times \llbracket \bar{t} \rrbracket_V \subseteq \mathbb{D} \setminus \Gamma_i \\ V, \forall^+(x, P) \vdash \forall x. \alpha & \quad \text{iff } \bigwedge (D_k, sp_k) \in P. \text{tp}(sp_k) = \text{tp}(P) \text{ and } V[x \mapsto D_k], sp_k \vdash \alpha \\ V, \exists^+(x, d, sp) \vdash \exists x. \alpha & \quad \text{iff } V[x \mapsto \{d\}], sp \vdash \alpha \end{aligned}$$

and dually for \exists^- and \forall^- . Here, $\llbracket \bar{t} \rrbracket_V$ represents a transformation of the list of values $\llbracket \bar{t} \rrbracket_v$ to the set of all possible lists of values generated by V . Set assignments allow us to delay deciding values for quantifier subproofs to the predicate base case. Note that $\{p\} \times \llbracket \bar{t} \rrbracket_V \subseteq \Gamma_i$ and $\{p\} \times \llbracket \bar{t} \rrbracket_V \subseteq \mathbb{D} \setminus \Gamma_i$ are decidable because due to our partitions, we only encounter finite and co-finite sets. The set-assignment-based validity check is thus executable and thus provides the algorithm that we use as our formally verified proof object checker: $v, p \vdash \alpha = (\lambda x. \{v(x)\}), p \vdash \alpha$ (proved by induction on α using Lemma 1).

4 Partitioned Decision Trees

Our proof system is parameterized with an assignment, but in our monitoring approach we are interested in computing a proof object for every assignment. In this section, we introduce *partitioned decision trees* (PDTs), a specialized data structure for representing and efficiently manipulating variable assignments, inspired by the use of BDDs in runtime verification [17]. We want to represent functions of the form $f : \mathbb{D} \times \dots \times \mathbb{D} \rightarrow \mathfrak{p}$, i.e., mappings from tuples of domain elements to proof trees, where each tuple corresponds to a variable assignment to the formula's free variables. As argued in the previous section, we are only interested in such functions with a finite range. Thus, we organize the domain into a finite number of subsets $\mathbb{D} \times \dots \times \mathbb{D}$ such that each tuple element is partitioned separately (using valued partitions over the domain). As before, we work with finite and co-finite sets in the partition. PDTs $\mathbb{P}(A)$ are defined inductively as follows:

$$\mathbb{P}(A) = \text{Leaf } A \mid \text{Node } (\mathbb{V}, \biguplus_{\mathbb{D}} (\mathbb{P}(A)))$$

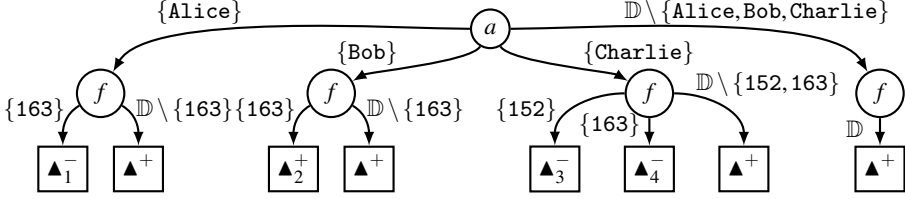


Fig. 6: Resulting PDT for our running example at time-point 3.

PDTs have leaves and nodes. Leaves store objects from the set A , while nodes store pairs of the form (x, P) , where x is a variable and P , a valued partition of the domain storing PDTs. PDTs generalize binary decision trees along two dimensions. First, the branching of their nodes is not binary but follows a given partition of the infinite domain \mathbb{D} . Second, their leaves do not store Boolean values. Instead, they store arbitrary objects, even though we will mostly use them with proof objects $A = p$. PDTs provide a way to organize the infinitely many possible variable assignments in a structured manner, storing only finitely many different proof objects. In monitoring, partitions will arise naturally, guided by the values occurring in the stream and assembled via operations that combine them.

Example 3. We continue the publish–approve example from Example 1. We consider the same stream but drop the top-level quantifiers from the formula φ : we only consider φ' with its free variables a and f . Figure 6 shows the PDT representing all assignments for φ' at time-point 3. The root node represents variable a , and the edges partition the values that a can take into the following domain subsets: $\{\text{Alice}\}$, $\{\text{Bob}\}$, $\{\text{Charlie}\}$, and $\mathbb{D} \setminus \{\text{Alice}, \text{Bob}, \text{Charlie}\}$. The second level is analogous for variable f . At every level of the PDT, the union of all choices cover the entire domain \mathbb{D} (by definition of partitions) and the partitions may differ at every node. The leaves of the PDT are different proof trees (formally, proof objects) which we represent by small black triangles. For example, Δ_3^- is the proof tree of φ' 's violation shown in Example 1. In contrast, Δ^+ (occurring in multiple leaves) is the proof tree shown in Figure 7 of φ' 's vacuous satisfaction: the left hand side of the implication ($\text{publish}(a, f)$) is violated for any assignment v updated by following the path from the PDT's root to the respective leaf (e.g., taking $a = \text{Alice}$ and $f = 42 \in \mathbb{D} \setminus \{163\}$). \square

$$\frac{\text{publish}(a, f) \notin \Gamma_3 \quad \frac{v, 3 \vdash^- \text{publish}(a, f) \quad P^-}{v, 3 \vdash^+ \varphi'} \rightarrow_L^+}{v, 3 \vdash^+ \varphi'}$$

Fig. 7: Proof tree Δ^+ .

Since PDTs are a generalization of BDDs, we use similar functions to manipulate them. We list the most important ones, for partitions and PDTs in Figure 8, but we only show and discuss the implementation of `apply2`, `merge2`, and `hide`. Most PDT-functions are parameterized by a variable list $vs :: \bar{V}$ fixing the variable order. The functions `map_part` and `apply1` lift unary functions on objects to partitions and PDTs respectively.

The functions `merge2` and `apply2` do the same for binary functions; `apply2` generalizes the well-known `apply` function on BDDs [16]. On leaves, `apply2` maps f to the objects. When operating on a leaf and a node, `apply2` pushes f partially applied to the leaf to the node's leaves using `apply1`. Finally, on pairs of nodes, it proceeds recursively depending which of x , y , and z are equal. The most interesting case, $x = y = z$ occurs when both PDTs partition the domain values for z in different ways. Thus, we must


```

map_part :: (A ⇒ B) ⇒  $\mathbb{U}_{\mathbb{D}}$ (A) ⇒  $\mathbb{U}_{\mathbb{D}}$ (B)
merge2 :: (A ⇒ B ⇒ C) ⇒  $\mathbb{U}_{\mathbb{D}}$ (A) ⇒  $\mathbb{U}_{\mathbb{D}}$ (B) ⇒  $\mathbb{U}_{\mathbb{D}}$ (C)
merge2 f [] P2 = []
merge2 f ((D1, v1) # P1) P2 =
    let P3 = map_filter (λ(D2, v2). if D1 ∩ D2 ≠ ∅ then Some (D1 ∩ D2, f v1 v2) else None) P2;
        P4 = map_filter (λ(D2, v2). if D2 \ D1 ≠ ∅ then Some (D2 \ D1, v2) else None) P2
    in P3 @ merge2 f P1 P4
pdt_of ::  $\bar{V}$  ⇒ A ⇒ A ⇒ 2( $\bar{V} \rightarrow \mathbb{D}$ ) ⇒  $\mathbb{P}$ (A) split_prod ::  $\mathbb{P}$ (A × B) ⇒  $\mathbb{P}$ (A) ×  $\mathbb{P}$ (B)
apply1 ::  $\bar{V}$  ⇒ (A ⇒ B) ⇒  $\mathbb{P}$ (A) ⇒  $\mathbb{P}$ (B) split_list ::  $\mathbb{P}(\bar{A})$  ⇒  $\mathbb{P}(\bar{A})$ 
apply2 ::  $\bar{V}$  ⇒ (A ⇒ B ⇒ C) ⇒  $\mathbb{P}$ (A) ⇒  $\mathbb{P}$ (B) ⇒  $\mathbb{P}$ (C)
apply2 vs f (Leaf l1) (Leaf l2) = Leaf (f l1 l2)
apply2 vs f (Leaf l1) (Node (x, P2)) = Node (x, map_part (apply1 vs (λl2. f l1 l2)) P2)
apply2 vs f (Node (x, P1)) (Leaf l2) = Node (x, map_part (apply1 vs (λl1. f l1 l2) P1)
apply2 (z # vs) f (Node (x, P1)) (Node (y, P2)) =
    if x = z and y = z then Node (z, merge2 (apply2 vs f) P1 P2)
    else if x = z then Node (x, map_part (λl. apply2 vs f l (Node (y, P2))) P1)
    else if y = z then Node (y, map_part (λr. apply2 vs f (Node (x, P1)) r) P2)
    else apply2 vs f (Node (x, P1)) (Node (y, P2))
apply3 ::  $\bar{V}$  ⇒ (A ⇒ B ⇒ C ⇒ D) ⇒  $\mathbb{P}$ (A) ⇒  $\mathbb{P}$ (B) ⇒  $\mathbb{P}$ (C) ⇒  $\mathbb{P}$ (D)
hide ::  $\bar{V}$  ⇒ (A ⇒ A) ⇒ ( $\mathbb{U}_{\mathbb{D}}$ (A) ⇒ A) ⇒  $\mathbb{P}$ (A) ⇒  $\mathbb{P}$ (A)
hide vs leaf node (Leaf l) = Leaf (leaf l)
hide [z] leaf node (Node (x, P)) = Leaf (node (map_part unleaf P))
hide (z # vs) leaf node (Node (x, P)) =
    if x = z then Node (z, map_part (hide vs leaf node) P) else hide vs leaf node (Node (x, P))
    
```

Fig. 8: Selected functions on partitions and PDTs.

combine both partitions. For this, we use `merge2` that takes two valued partitions P_1 and P_2 , and iteratively “erodes” P_2 by intersecting its elements with the sets in P_1 while applying f . Since both P_1 and P_2 cover \mathbb{D} , the resulting set of intersections is a valued partition. The function `apply3` analogously combines three PDTs into one.

The function `hide` traverses the PDT similarly to `apply1`, while eliminating the last variable in the given variable list. It uses two higher-order arguments, in case the last layer is present (*node*) or absent (*leaf*). The function `pdt_of vs A B V` constructs a PDT from a finite set of partial assignments ($V :: 2^{(\bar{V} \rightarrow \mathbb{D})}$) using A for leaves reached by paths from the set, and B for the other leaves. Finally, the `split_*` functions transpose a PDT storing pairs (lists of equal length) into a pair (list) of PDTs.

5 Monitoring Algorithm

We follow the typical online monitoring algorithm structure consisting of an initialization and a step (evaluation) function [25, 30]. The initializer `init` (omitted as standard) computes our monitor’s initial state $s \in \mathbb{S}$ from an MFOTL specification α . Figure 9 shows an excerpt of our monitor’s state, which recursively follows the formula structure and augments some operators with additional information, such as buffers storing verdicts from subformulas (\mathbb{B}_2 for \wedge and \mathbb{B}_3 for \mathcal{S}) or an operator-specific state (\mathbb{S}_{saux} for \mathcal{S}).

$$\begin{aligned}
\mathbb{B}_2 &= \overline{\mathbb{P}(\mathbf{p})} \times \overline{\mathbb{P}(\mathbf{p})} & \mathbb{B}_3 &= \overline{\mathbb{P}(\mathbf{p})} \times \overline{\mathbb{P}(\mathbf{p})} \times \overline{\mathbb{N}} \times \overline{\mathbb{N}} & \mathbb{S}_{saux} &= \dots \\
\mathbb{S} &= \text{MPred } \mathbb{E} \, \mathbb{T} \mid \text{MAnd } \mathbb{S} \, \mathbb{S} \, \mathbb{B}_2 \mid \text{MExists } \mathbb{E} \, \mathbb{S} \mid \text{MSince } \mathbb{I} \, \mathbb{S} \, \mathbb{S} \, \mathbb{B}_3 \, (\overline{\mathbb{P}(\mathbb{S}_{saux})}) \mid \dots \\
\text{eval} &:: \overline{\mathbb{V}} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{E} \times \overline{\mathbb{D}} \Rightarrow \mathbb{S} \Rightarrow \overline{\mathbb{P}(\mathbf{p})} \times \mathbb{S} \\
\text{eval } vs \, \tau \, i \, \Gamma \, (\text{MPred } p \, ts) &= \\
&\quad \text{let } e = \text{pdt_of} \, (\text{filter } (\lambda v. v \in \text{fv}(ts)) \, vs) \, (p^+(i, p, ts)) \, (p^-(i, p, ts)) \\
&\quad \quad \{ \sigma \mid \exists ds. p(ds) \in \Gamma \wedge \text{match } ts \, ds = \text{Some } \sigma \} \text{ in } ([e], \text{MPred } p \, ts) \\
\text{eval } vs \, \tau \, i \, \Gamma \, (\text{MAnd } s_1 \, s_2 \, buf) &= \text{let } (es_1, s'_1) = \text{eval } vs \, \tau \, i \, \Gamma \, s_1; \quad (es_2, s'_2) = \text{eval } vs \, \tau \, i \, \Gamma \, s_2; \\
&\quad (es, buf') = \text{buf2_take} \, (\text{apply2 } vs \, \text{do_and}) \, (\text{buf2_add } buf \, es_1 \, es_2) \text{ in } (es, \text{MAnd } s'_1 \, s'_2 \, buf') \\
\text{eval } vs \, \tau \, i \, \Gamma \, (\text{MExists } x \, s) &= \text{let } (es, s') = \text{eval } (vs@[x]) \, \tau \, i \, \Gamma \, s \\
&\quad \text{in } (\text{map} \, (\text{hide } (vs@[x]) \, (\text{do_exists_leaf } x) \, (\text{do_exists_node } x)) \, es, \text{MExists } x \, s') \\
\text{eval } vs \, \tau \, i \, \Gamma \, (\text{MSince } I \, s_1 \, s_2 \, buf \, saux) &= \\
&\quad \text{let } (es_1, s'_1) = \text{eval } vs \, \tau \, i \, \Gamma \, s_1; \quad (es_2, s'_2) = \text{eval } vs \, \tau \, i \, \Gamma \, s_2; \\
&\quad (es, buf', saux') = \text{buf2t_take} \, (\lambda e_1 \, e_2 \, (\tau, i) \, saux). \\
&\quad \quad \text{let } (saux', es') = \text{split_prod} \, (\text{apply3 } vs \, (\text{update_since } I \, \tau \, i) \, e_1 \, e_2 \, saux') \\
&\quad \quad \text{in } (saux', \text{split_list } es') \, (\text{buf2t_add } buf \, es_1 \, es_2 \, [(\tau, i)]) \, saux' \\
&\quad \text{in } (es, \text{MSince } I \, s'_1 \, s'_2 \, buf' \, saux')
\end{aligned}$$

Fig. 9: Involved types and selected cases of the monitor's eval function

$$\begin{aligned}
\text{do_exists_leaf } x \, p &= \text{if } p \in \mathbb{sp} \text{ then } \exists^+(x, d \leftarrow \mathbb{D}, p) \text{ else } \exists^-(x, [(\mathbb{D}, p)]) \\
\text{do_exists_node } x \, P &= \text{if } \bigvee (D_i, p) \in P. p \in \mathbb{sp} \\
&\quad \text{then min } (\text{map_filter } (\lambda (D_i, p). \text{if } p \in \mathbb{sp} \text{ then } \text{Some } (\exists^+(x, d \leftarrow D_i, p))) \text{ else None}) \, P) \\
&\quad \text{else } \exists^-(x, P)
\end{aligned}$$

Fig. 10: Functions do_exists_leaf and do_exists_node.

Our function eval, partly shown in Figure 9, takes as inputs a new time-point i (along with its time-stamp τ and database Γ) and a monitor state s and outputs the next state s' and a list of PDTs of proof objects as verdicts. (In addition, eval keeps track of the variable ordering used in PDTs via the parameter vs .) Lists in the output are necessary because delays may occur for (bounded) future operators and a single time-point might trigger multiple outputs. Our algorithm extends Lima et al.'s algorithm [25] computing proof trees for MTL. We highlight our key additions to eval and the state Figure 9 in gray.

We focus on the predicate, conjunction, existential quantifier, and since cases. In the predicate case, we find all partial assignments σ mapping the predicate's variables to the values ds , so that $p(ds) \in \Gamma$. We reuse VeriMon's match function [30] to compute such partial assignments. We convert this set of assignments to a PDT using `pdt_of`. In the resulting PDT, matching assignments lead to leaves using the satisfaction proof $p^+(i, p, ts)$, whereas the others lead to the corresponding violation proof $p^-(i, p, ts)$.

The conjunction case is taken almost without changes from Lima et al.'s [25] MTL algorithm. We reuse the buffering functions `buf2_add` and `buf2_take`. The first adds partial results to the buffer, while the second combines these results and dequeues them once both subformulas have produced results for a time-point. The only difference is that our buffers store PDTs of proof objects, whereas the MTL algorithm works with propositional proof objects. Accordingly, we reuse the Lima et al.'s function `do_and` :: $p \Rightarrow p \Rightarrow p$ to combine two proof objects conjunctively, but lift it to PDTs using `apply2`.

The quantifier cases are a new addition of our work. As both cases proceed dually, we focus on $\exists x. \alpha$ formulas. Considering that α may have one more free variable than $\exists x. \alpha$, the recursive call appends x to the variable list ordering. The recursive call's output

is processed using our function `hide` to eliminate the quantified variable. The interesting cases occur near the leaves of α 's PDTs. If x is not present, `hide` will encounter a leaf, i.e., a proof object, and use the function `do_exists_leaf` (Figure 10) to perform a case distinction: satisfactions result in a satisfaction (\mathfrak{sp}) of $\exists x.\alpha$ with an arbitrary element d of the domain as the witness (we write $x \leftarrow X$ to denote an arbitrarily chosen element x of a non-empty set X); violations result in a violation (\mathfrak{vp}) with the trivial partition. If x is present as the last decision node, then `hide` will use `do_exists_node` (Figure 10) to construct the proof object for $\exists x.\alpha$. It performs a case distinction whether a satisfaction proof is contained in the partition of this last node. If it is, $\exists x.\alpha$ is satisfied and we compute the smallest (in proof size) such satisfaction proof, taking as our witness an arbitrary element of the respective partition set. Otherwise, all leaves are violations and we obtain a violation proof of $\exists x.\alpha$.

To reuse Lima et al.'s [25] temporal operator evaluation, our state stores a PDT whose leaves are the auxiliary state of these algorithms (instead of proof objects). This allows us to keep the complex auxiliary state and its update unchanged. For example, we use `apply3` to lift Lima et al.'s [25] `update_since` function to two PDTs storing proof objects for subformulas and a third one storing the auxiliary state. The resulting PDT has type $\mathbb{P}(\mathbb{S}_{saux} \times \bar{p})$, which we transpose into the desired $\mathbb{P}(\mathbb{S}_{saux}) \times \mathbb{P}(\bar{p})$ using `split_prod` and `split_list`.

6 Implementation and Case Study

We implement our algorithm in a new monitoring tool, called WHYMON [2]. Our implementation consists of 4500 lines of OCaml code and incorporates an optimization of collapsing partition sets with the same stored values both in proof objects and in PDTs. Our formally verified checker contributes additional 1700 lines of OCaml code generated from our Isabelle formalization, which itself comprises 6400 lines of definitions and proofs. The checker's main function lifts the validity check of proof objects (\vdash) to PDTs, i.e., `check : trace \rightarrow formula \rightarrow pdt \rightarrow bool`, and is used to certify WHYMON's output. WHYMON includes a visualization [3] implemented in React [20] that consists of 2400 lines of JavaScript and invokes a JavaScript version of our monitor, generated by `Js_of_ocaml` [32]. Here, we consider the data race policy [18] that captures possible concurrency issues in multithreaded programs on a stream prefix generated by Raszyk [27, Section 4.3]. Furthermore, we consider Nokia's Data-collection Campaign [4], which comes with a stream prefix of around 5 million time-points [1], for which we focus on the *del-2-3* policy [12] controlling data propagation between databases. We describe a violation for each scenario highlighting the advantages of our approach.

Example 4. We first return to Example 3 in our visualization tool, depicted in Figure 11. The table includes TP (time-points), TS (time-stamps), and Values columns. The following columns show the topmost operator of φ' 's subformulas or its predicate names (and their variables). In the Values column, for each of the already evaluated time-points, there is an associated button enclosing a \checkmark (for satisfactions), or a \times (for violations) or both. After clicking on this button, we are presented with a dropdown menu (as in Figure 12) that corresponds to a partition. The listed values are the (potentially multiple) variable assignments of the resulting PDT for that specific time-point. The formula φ' contains two free variables, a and f , and to single out a verdict we must select one value for

each. In particular, at time-point 3 we select $a = \text{Charlie}$ and $f = 152$. Note that in the visualization we focus on readability and omit set parentheses. Moreover, **Other** denotes the complement of the listed values. After choosing the assignments, a Boolean verdict appears in the next column matching the topmost operator of φ' , namely \rightarrow . Clicking on this Boolean verdict reveals and highlights the Boolean verdicts associated with its justification. The subformulas' columns of the current inspection are also highlighted. In this case, the implication is violated because the left side is satisfied, while the $\blacklozenge_{[0,7]}$ subformula is violated. We can explore this verdict further: the violation is justified by those of its subformula at time-points 2 and 3 (the time-points inside the interval are also highlighted). For each time-point, there is another dropdown menu where we can select an assignment for m . Here, the only listed value is **Any**, which corresponds to \mathbb{D} . Thus, the existential quantifier is violated because the subformula $\text{approve}(m, 152)$ is violated for all values that m can be assigned to (\mathbb{D}), and all justifications are identical.

Data Race Detection Multithreaded programs are pervasive and hard to debug. In particular, they are prone to data races, which occur when two threads access (read or write to) a shared address concurrently and at least one of these accesses is a write. Locking mechanisms that synchronize access to variables shared between threads are a plausible solution. We consider the following policy to detect data race potentials [18]:

$$\begin{aligned} \varphi_{dr} &= \text{data race}(t_1, t_2, x) \rightarrow \exists l. (\text{acq nrel}(t_1, x, l) \wedge \text{acq nrel}(t_2, x, l)), \text{ with} \\ \text{data race}(t_1, t_2, x) &= \blacklozenge (\text{read}(t_1, x) \vee \text{write}(t_1, x)) \wedge \blacklozenge \text{write}(t_2, x), \text{ and} \\ \text{acq nrel}(t, x, l) &= \blacksquare ((\text{read}(t, x) \vee \text{write}(t, x)) \rightarrow (\neg \text{rel}(t, l) \mathcal{S} \text{acq}(t, l))) \end{aligned}$$

where the predicates $\text{read}(t, x)$ and $\text{write}(t, x)$ specify read and write operations performed by thread t to shared address x , and $\text{acq}(t, l)$ and $\text{rel}(t, l)$ specify the acquisition and the release of lock l by thread t . Havelund et al. [18] consider a closed formula variant of this policy as their tool, DEJAVU, only supports closed formulas. In contrast, WHYMON supports open formulas. We consider the stream prefix:

$$\begin{aligned} & \langle (0, \{\text{acq}(9, 9)\}), (1, \{\text{read}(9, 3)\}), (2, \{\text{acq}(13, 19)\}), (3, \{\text{acq}(15, 3)\}), \\ & (4, \{\text{acq}(18, 15)\}), (5, \{\text{read}(13, 5)\}), (6, \{\text{write}(15, 4)\}), (7, \{\text{write}(15, 3)\}), \dots \rangle \end{aligned}$$

At time-point 7, WHYMON outputs a PDT with non-trivial assignments. We focus on the single violation in this PDT, which corresponds to the assignment $(\{9\}, \{15\}, \{3\})$ for (t_1, t_2, x) . This violation is shown in Figure 13. The topmost operator of φ_{dr} is an implication, and it is violated because the left side is satisfied (there was a data race), while the right side (the lock requirement) is violated. Specifically, the data race occurred because thread $t_1 = 9$ read address 3 at time-point 1, satisfying the $\blacklozenge_{[0, \infty]}$ subformula in the left conjunct, and thread $t_2 = 15$ wrote to address 3 at the current time-point 7, satisfying the $\blacklozenge_{[0, \infty]}$ subformula in the right conjunct. Moving to the right side of the implication, the violation of the existential indicates that its subformula is violated for every value of \mathbb{D} . In particular, the subsets of the domain $\{9\}$ and $\{9\}^c$ are each associated with a different violation. Here, we focus on the violation where $l = 9$. The subformula is a conjunction, and to be violated it suffices that one of the conjuncts is violated. This violation stems from the violation of the right conjunct $\blacksquare_{[0, \infty]}$ (note that $t_2 = 15$ is listed as the variable in the predicate columns). We omit the columns referring to the left conjunct, since all

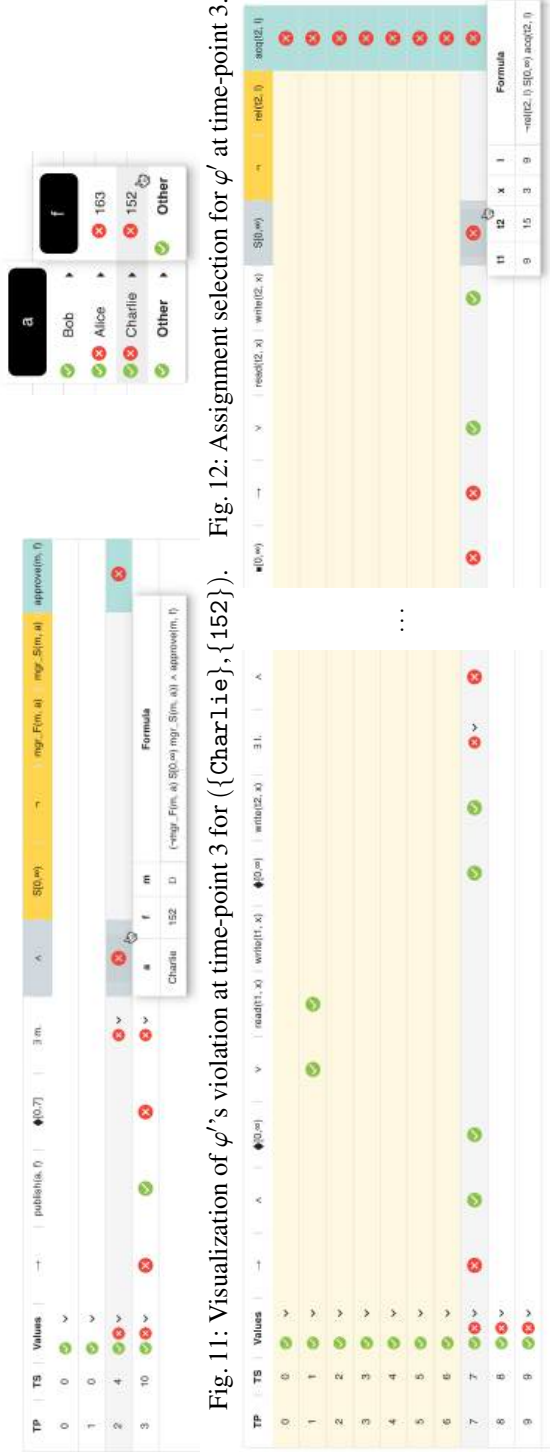


Fig. 12: Assignment selection for ϕ' at time-point 3.

Fig. 13: Visualization of ϕ_{dr} 's violation at time-point 7 for $(\{9\}, \{15\}, \{3\})$.

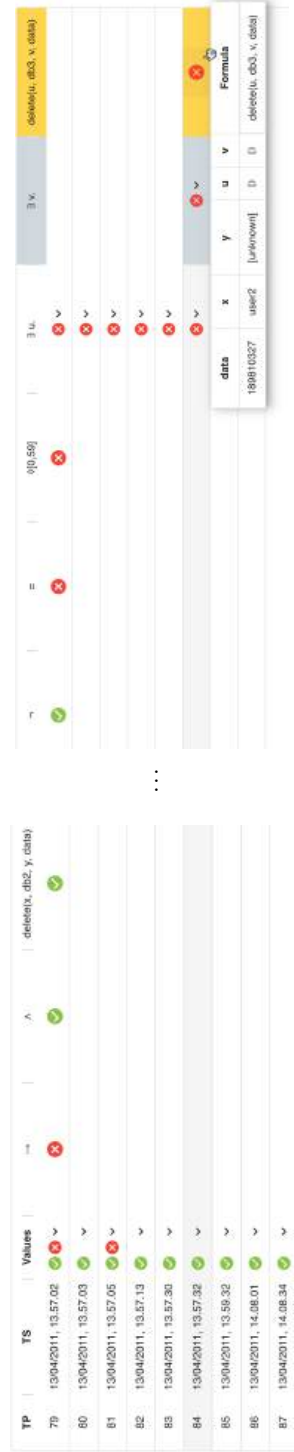


Fig. 14: Visualization of ϕ_{del} 's violation at time-point 79 for $(\{189810327\}, \{user2\}, \{[unknown]\})$.

entries are empty. Once again, the implication is violated because the left side is satisfied, i.e., thread $t_2 = 15$ wrote to address 3 at time-point 7, satisfying the disjunction, but $S_{[0,\infty)}$ on the right side is violated, because thread $t_2 = 15$ never acquired the lock $l = 9$.

Data Propagation Nokia’s Data-collection Campaign [4] used three databases db_1 , db_2 and db_3 in the collection of sensitive information from mobile phones of participants. We focus on the policy φ_{del} [12], which controls the data propagation between databases db_2 and db_3 : if *data* is deleted from db_2 , then it must be deleted from db_3 within 1 minute.

$$\varphi_{del} = \text{delete}(x, db_2, y, data) \wedge data \not\approx [\text{unknown}] \rightarrow \Diamond_{[0,60]} \exists u, v. \text{delete}(u, db_3, v, data)$$

where db_2 , db_3 , and $[\text{unknown}]$ are constants and $\text{delete}(db_{user}, db, p_{id}, data)$ specifies the deletion of *data* from participant p_{id} from database db using database user db_{user} . We used the REPLAYER tool [22] to convert the stream prefix to WHYMON’s format. We executed WHYMON’s command line interface with the entire prefix and found two violations. The following experiments were conducted on a computer with an Apple M1 Chip (8 cores) and 16GB of RAM. WHYMON took 17m51s to process the entire prefix. We also executed MONPOLY with a slightly modified yet equivalent policy (due to monitorability restrictions), and its running time amounted to 1m10s. MONPOLY outperforms WHYMON, but we must acknowledge the different outputs both monitors produce. MONPOLY only outputs variable assignments, whereas WHYMON outputs entire PDTs containing all assignments and a justification of the verdict in the form of a proof tree for each. We extract 100 time-points containing both violations and focus on the violation at time-point 79 for the assignment $(\{189810327\}, \{\text{user2}\}, \{[\text{unknown}]\})$ for $(data, x, y)$, depicted in Figure 14. Time-stamps are converted to actual dates (by enabling the option) and we omit time-points that do not contain relevant events for the violation. Let

$$\begin{aligned} \Gamma_{79} &= \{\text{delete}(\text{user2}, db_2, [\text{unknown}], 189810327), \\ \Gamma_{80} &= \{\text{delete}(\text{triggers}, db_3, [\text{unknown}], [\text{unknown}])\}, \\ \Gamma_{81} &= \{\text{delete}(\text{user2}, db_2, [\text{unknown}], 189810328)\}, \text{ and } \Gamma_{82} = \Gamma_{83} = \Gamma_{84} = \emptyset. \end{aligned}$$

The implication is violated because the left side is satisfied (there was a deletion at the current time-point 79), but $\Diamond_{[0,59]}$ is violated. Note that $[0, 60]$ was replaced with the equivalent interval $[0, 59]$. For each time-point of $[E_{79}^f([0, 59]), L_{79}^f([0, 59])] = \{79, \dots, 84\}$, the subformula is violated. Regardless of the values we assign to u and v (all violations are identical), the subformula $\text{delete}(u, db_3, v, 189810327)$ is violated.

7 Conclusion

We describe an approach for MFOTL monitoring with verdicts in the form of proof objects for every free variable assignment. Such verdicts are useful for understandability and certification, which increases the monitor’s trustworthiness. We implement our approach in the tool WHYMON along with an interactive visualization for these verdicts, which we invite the reader to explore [3]. As future work, we plan to provide support for equality between variables and to improve our monitor’s performance by, e.g., stream slicing [29].

Data Availability Statement Our artifact [26] includes WHYMON’s source code at the artifact submission time together with instructions on how to set up WHYMON locally, extract our PDT checker, execute our examples, and replicate our case study.

Acknowledgements This research is supported by a Novo Nordisk Fonden start package grant (NNF20OC0063462). We thank David Basin, François Hublet, Srđan Krstić, Matthias Lott, Joshua Schneider for their suggestions on WHYMON's and EXPLANATOR2's user interfaces. We are also grateful to anonymous TACAS 2024 reviewers, who helped us improve the presentation of this paper with their valuable comments.

References

1. The Nokia case study log file (2014), <https://sourceforge.net/projects/monpoly/files/ldcc.tar/download>
2. WHYMON repository (2023), <https://github.com/runtime-monitoring/whymon>
3. WHYMON web interface (2023), <https://runtime-monitoring.github.io/whymon>
4. Aad, I., Niemi, V.: NRC data collection campaign and the privacy by design principles. In: Proceedings of the International Workshop on Sensing for App Phones (PhoneSense) (2010)
5. Ailamazyan, A.K., Gilula, M.M., Stolboushkin, A.P., Schwartz, G.F.: Reduction of a relational model with infinite domains to the case of finite domains. *Doklady Akademii Nauk SSSR* **286**(2), 308–311 (1986), <http://mi.mathnet.ru/dan47310>
6. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. *Inf. Comput.* **104**(1), 35–77 (1993). <https://doi.org/10.1006/inco.1993.1025>
7. Arfelt, E., Basin, D.A., Debois, S.: Monitoring the GDPR. In: Sako, K., Schneider, S.A., Ryan, P.Y.A. (eds.) ESORICS 2019. LNCS, vol. 11735, pp. 681–699. Springer (2019). https://doi.org/10.1007/978-3-030-29959-0_33
8. Basin, D.A., Bhatt, B.N., Krstić, S., Traytel, D.: Almost event-rate independent monitoring. *Formal Methods Syst. Des.* **54**(3), 449–478 (2019). <https://doi.org/10.1007/s10703-018-00328-3>
9. Basin, D.A., Bhatt, B.N., Traytel, D.: Optimal proofs for linear temporal logic on lasso words. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 37–55. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_3
10. Basin, D.A., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. *Formal Methods Syst. Des.* **49**(1-2), 75–108 (2016). <https://doi.org/10.1007/s10703-016-0242-y>
11. Basin, D.A., Dietiker, D.S., Krstić, S., Pignolet, Y., Raszyk, M., Schneider, J., Ter-Gabrielyan, A.: Monitoring the internet computer. In: Chechik, M., Katoen, J., Leucker, M. (eds.) FM 2023. LNCS, vol. 14000, pp. 383–402. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_22
12. Basin, D.A., Harvan, M., Klaedtke, F., Zalinescu, E.: Monitoring data usage in distributed systems. *IEEE Trans. Software Eng.* **39**(10), 1403–1426 (2013). <https://doi.org/10.1109/TSE.2013.18>
13. Basin, D.A., Klaedtke, F., Müller, S.: Monitoring security policies with metric first-order temporal logic. In: Joshi, J.B.D., Carminati, B. (eds.) SACMAT 2010. pp. 23–34. ACM (2010). <https://doi.org/10.1145/1809842.1809849>
14. Basin, D.A., Klaedtke, F., Zalinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 15:1–15:45 (2015). <https://doi.org/10.1145/2699444>
15. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017). <https://doi.org/10.29007/89hs>
16. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* **35**(8), 677–691 (1986). <https://doi.org/10.1109/TC.1986.1676819>

17. Havelund, K., Peled, D.: BDDs for representing data in runtime verification. In: Deshmukh, J., Nickovic, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 107–128. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_6
18. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. *Formal Methods Syst. Des.* **56**(1), 1–21 (2020). <https://doi.org/10.1007/s10703-018-00327-4>
19. Hull, R., Su, J.: Domain independence and the relational calculus. *Acta Informatica* **31**(6), 513–524 (1994). <https://doi.org/10.1007/BF01213204>
20. Hunt, P., O’Shannessy, P., Smith, D., Coatta, T.: React: Facebook’s functional turn on writing JavaScript. *ACM Queue* **14**(4), 40 (2016). <https://doi.org/10.1145/2984629.2994373>
21. Koymans, R.: Specifying real-time properties with metric temporal logic. *Real Time Syst.* **2**(4), 255–299 (1990). <https://doi.org/10.1007/BF01995674>
22. Krstić, S., Schneider, J.: A benchmark generator for online first-order monitoring. In: Deshmukh, J., Nickovic, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 482–494. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_27
23. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. *Int. J. Softw. Tools Technol. Transf.* **4**(2), 224–233 (2003). <https://doi.org/10.1007/s100090100062>
24. Leucker, M., Sánchez, C., Scheffell, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) SAC 2018. pp. 1925–1933. ACM (2018). <https://doi.org/10.1145/3167132.3167338>
25. Lima, L., Herasimau, A., Raszyk, M., Traytel, D., Yuan, S.: Explainable online monitoring of metric temporal logic. In: TACAS 2023. LNCS, vol. 13994, pp. 473–491. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_28
26. Lima, L., Huerta y Munive, J.J., Traytel, D.: Artifact for "Explainable online monitoring of metric first-order temporal logic" (2024). <https://doi.org/10.5281/zenodo.10439544>
27. Raszyk, M.: Efficient, Expressive, and Verified Temporal Query Evaluation. Ph.D. thesis, ETH Zürich (2022). <https://doi.org/10.3929/ethz-b-000553221>
28. Raszyk, M., Basin, D.A., Krstić, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Chen, Y., Cheng, C., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 151–170. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_9
29. Schneider, J., Basin, D.A., Brix, F., Krstić, S., Traytel, D.: Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.* **23**(2), 185–208 (2021). <https://doi.org/10.1007/s10009-021-00607-1>
30. Schneider, J., Basin, D.A., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_18
31. Ulus, D.: Online monitoring of metric temporal logic using sequential networks. *CoRR abs/1901.00175* (2019). <https://doi.org/10.48550/arxiv.1901.00175>
32. Vouillon, J., Balat, V.: From bytecode to JavaScript: the Js_of_ocaml compiler. *Softw. Pract. Exp.* **44**(8), 951–972 (2014). <https://doi.org/10.1002/spe.2187>

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.



Proof Checking



IsaRARE: Automatic Verification of SMT Rewrites in Isabelle/HOL

Hanna Lachnitt¹(✉) , Mathias Fleury⁴ , Leni Aniva¹ ,
Andrew Reynolds² , Haniel Barbosa³ , Andres Nötzli⁵ , Clark Barrett¹ ,
and Cesare Tinelli² 

¹ Stanford University, Stanford, USA
lachnitt@stanford.edu

² The University of Iowa, Iowa City, USA

³ Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

⁴ University of Freiburg, Freiburg, Germany

⁵ Cubist, Inc., San Diego, USA

Abstract. Satisfiability modulo theories (SMT) solvers are widely used to ensure the correctness of safety- and security-critical applications. Therefore, being able to trust a solver’s results is crucial. One way to increase trust is to generate independently checkable proof certificates, which record the reasoning steps done by the solver. A key challenge with this approach is that it is difficult to efficiently and accurately produce proofs for reasoning steps involving term rewriting rules. Previous work showed how a domain-specific language, RARE, can be used to capture rewriting rules for the purposes of proof production. However, in that work, the RARE rules had to be trusted, as the correctness of the rules themselves was not checked by the proof checker. In this paper, we present IsaRARE, a tool that can automatically translate RARE rules into Isabelle/HOL lemmas. The soundness of the rules can then be verified by proving the lemmas. Because an incorrect rule can put the entire soundness of a proof system in jeopardy, our solution closes an important gap in the trustworthiness of SMT proof certificates. The same tool also provides a necessary component for enabling full proof reconstruction of SMT proof certificates in Isabelle/HOL. We evaluate our approach by verifying an extensive set of rewrite rules used by the cvc5 SMT solver.

1 Introduction

Satisfiability modulo theories (SMT) [8] solvers provide the back-end reasoning power for many formal methods applications. These applications are often used to provide safety or security guarantees for critical systems [1, 15, 21, 23]. For such applications, an incorrect result from a solver could have catastrophic consequences. Thus, ensuring the correctness of a solver’s results is crucial. However, industrial-strength SMT solvers are large and complex software systems which are under constant active development. As with any other large software project,

* This work was supported in part by the Stanford Center for Automated Reasoning and by a gift from Amazon Web Services.

even when employing software engineering best practices, it is unrealistic to expect that solvers do not contain implementation bugs that could, in the worst case, compromise the correctness of their answers.

One solution is to formally verify the SMT solver itself. Unfortunately, that would be a massive effort. It would likely require performance compromises [17] and impose a tremendous maintenance burden, as changes to solvers are frequent, and each change would require revisiting the verification.

Fortunately, there is a less expensive alternative: we can independently check each result produced by a solver. This is generally easy when the result is “satisfiable,” at least for quantifier-free inputs. The solver can produce a model and we can check via evaluation that the input formula indeed holds in it. To have a similar ability to check a result of “unsatisfiable,” solvers must be instrumented to produce *proof certificates* that can be independently verified by a separate proof checker. To maximize trustworthiness, the proof checker should be small, simple, and, ideally, formally verified. Alternatively, the checker can be embedded in a highly trusted system such as a skeptical interactive theorem prover. The SMT community is increasingly embracing proof production, with it becoming a major focus in recent years [3, 4, 19, 29].

One of the main challenges faced by SMT proof production efforts is the extensive use of theory-specific *term rewriting rules*. There are hundreds of such rules in modern solvers, each of which must be justifiable using some proof rule. Nötzli et al. [28] introduced a methodology for producing proofs for term rewriting rules by using the RARE domain-specific language. In that work, rules are defined in RARE, imported by a solver, and then used to elaborate the solver’s term rewriting proof steps into finer-grained proofs using the RARE rules. This approach has proved to be viable in the CVC5 SMT solver [2]. However, previous work did not address the *correctness* of the rules, i.e., it does not ensure that an incorrect RARE rule does not compromise the correctness of proof certificates.

An incorrect rule can have severe consequences. First of all, it may affect the ability of the solver to produce a proof certificate at all: if the incorrect rule does not match what the solver code does, then the elaboration of the term rewriting proof steps with RARE may fail. More concerning, if both the code and the proof rule are incorrect in the same way (perhaps because one was modeled after the other), then proof elaboration may succeed, but the proof certificate will be incorrect because it uses an invalid rule. This is especially problematic when using proof checkers that consider proof rules as trusted—that is, they only check whether rules are applied correctly and do not check the rules themselves.

There are two ways to fill this gap. One is to separately verify the proof rules; another is to use a more sophisticated proof checker, for example, one embedded in a skeptical interactive theorem prover, that will fail if an invalid rule is used. In this paper, we introduce IsaRARE, a new plugin for the Isabelle/HOL proof assistant [27] (abbreviated to just Isabelle going forward), which can do the former and is a necessary step towards the latter. The plugin translates RARE rules into the language of Isabelle where they can then be formally proved as lemmas. Note that when using IsaRARE simply as a rewrite rule verifier, the translation

from RARE to Isabelle becomes another trusted component. We mitigate this by reusing extensively-tested infrastructure in Isabelle for the translation.

To show the effectiveness of IsaRARE, we implemented a large number of new rules in RARE (beyond those in [28]) needed to elaborate term rewriting steps in proofs generated by the CVC5 SMT solver [2]. We show that IsaRARE can translate all of these rules into corresponding lemmas in Isabelle and can prove the majority of them automatically. In ongoing work, we are manually providing proofs for the rest, and have already proven most of them.

Our long-term vision is to enable the full integration of CVC5 and Isabelle via proof certificate reconstruction. Currently, Isabelle can send proof obligations to CVC5, but it is unable to automatically reconstruct Isabelle proofs from CVC5’s proof certificates. Our goal is to enable Isabelle to reconstruct *every* step in these proof certificates. In order to reach this goal, it is essential to have rewrite lemmas available for reconstructing rewrite steps, as they appear in almost all proofs, and without dedicated support for discharging rewrite proof steps, reconstruction in Isabelle can fail [11, 31].

In summary, we make the following contributions:

- we introduce IsaRARE, an Isabelle plugin for generating correctness lemmas for RARE rules;
- we add several new features to RARE itself and implement 163 new rewrite rules in RARE, almost tripling the size of the rule database from [28];
- we evaluate IsaRARE, showing that it can translate all of the RARE rules into Isabelle lemmas and can prove the majority of them automatically.

In the rest of the paper, after surveying related work, we give an overview of proof production and the interface to Isabelle (Section 2). Then, we present the RARE language and our extensions (Section 3). We next introduce IsaRARE and explain the challenges in transforming a RARE rule to an Isabelle lemma (Section 4). Finally, we present an evaluation of our approach (Section 5).

1.1 Related Work

Various attempts at proof production in SMT solvers have been implemented in the past [7, 13, 14, 22, 25], though these implementations typically either produce proofs certificates that are too coarse-grained (that is, they do not provide enough information for efficient proof checking) or produce them only if critical components are disabled, making solving while producing proofs slow or incomplete. Producing complete, independently-checkable proofs remains challenging.

One major challenge is solved by the modular framework by Barbosa et al. [3]. It enables proof production during term rewriting and formula processing and has been implemented in the SMT solver veriT [13] using the Alethe proof format [32]. Hoenicke and Schindler [19] introduce an alternative approach, implemented in the solver SMTInterpol [14], which also allows proof production for term rewriting and formula processing. Both of these approaches assume that the set of rewrite rules that can be used in proofs is fixed. Their sets include rules

for rewriting over equality, rules for rewriting Boolean formulas, and rules for reasoning about arithmetic. Notably absent, however, are rules for string and bit-vector rewrites. In other work, Barbosa et al. [4] describe a general architecture where the only holes in the generated proof certificates are those from rewrite steps. One of their key ideas is to support lazy proof production via a post-processing proof reconstruction step. This capability is leveraged in the work by Nötzli et al. [28] to produce proofs for rewrite steps based on rules written in RARE, which is the starting point for this work.

The interactive theorem prover Isabelle [30] includes a popular tool called Sledgehammer [9], which encodes proof obligations as SMT problems and uses SMT solvers to solve them. Sledgehammer currently supports proof reconstruction [12, 18] for two SMT solvers: z3 [26] and veriT [13]. However, z3 provides only coarse-grained proofs, which can cause reconstruction to fail. This issue was addressed for veriT by manually translating and proving correct in Isabelle the predefined set of rewrite rules in Alethe [18, 31]. Our work improves on this effort by providing an *automatic* mechanism for translating an *extendable* set of rewrite rules into Isabelle and includes support for bit-vector and string rewrites unsupported by veriT.

2 Preliminaries

2.1 Satisfiability Modulo Theories (SMT)

The underlying logic of SMT is many-sorted first-order logic with equality (see e.g., [16]). A *signature* Σ consists of a set $\Sigma^s \subseteq S$ of sort symbols and a set Σ^f of sorted function symbols with sorts from Σ^s . We assume the usual definitions of well-sorted terms, literals, and formulas. We also use the usual definition of interpretations and of a satisfiability relation \models between Σ -interpretations and Σ -formulas. A Σ -theory T is a non-empty class of Σ -interpretations closed under variable reassignment. A Σ -formula φ is *T-satisfiable* (resp., *T-unsatisfiable*, *T-valid*) if it is satisfied by some (resp., no, all) interpretation(s) in T . For the rest of the paper, we assume (un)satisfiability is always with respect to some given background theory T .

2.2 SMT Proofs and Rewriting

A *proof (of unsatisfiability)* is a series of inference steps starting from an input formula and terminating with \perp , showing that the input formula is unsatisfiable. The *granularity* of a proof step refers to how much reasoning it requires and roughly corresponds to the complexity of checking that the step is correct. In particular, steps (and thus the proofs containing them) are *fine-grained* if they can be efficiently checked, and *coarse-grained* otherwise. We will often refer to coarse-grained steps as *holes*.

One approach for the efficient production of proofs is to introduce coarse-grained proof steps for certain performance-critical deductions made while solv-

ing and then go back and fill in these holes with fine-grained steps as a post-processing step. We refer to this as *proof elaboration*, and it is particularly appealing for rewriting steps, since SMT solvers have hundreds of different rewrites to simplify and normalize terms, and instrumenting the rewriting code to produce fine-grained proofs is difficult and may introduce an unacceptable degradation in performance.

The approach taken by Nötzli et al. [28], and the one we also follow in this paper, is to assume that the SMT solver uses *generic* proof steps for all rewrites during solving and then elaborates these steps during post-processing by consulting a database of specific rewrite rules. The database is constructed by defining a set of rewrite rules in the domain-specific language RARE, which we discuss in Section 3. The elaboration tries to find one or more rules from the database to justify each generic, coarse-grained rewrite step. Additionally, it uses a built-in *evaluate* rule to justify steps that hold purely via constant folding. If elaboration is successful, the generic step is replaced by the fine-grained steps from the database.

2.3 SMT in Isabelle

As mentioned above, Sledgehammer [9] is an Isabelle tactic that applies automated reasoning tools, including SMT solvers, to prove goals in Isabelle. When targeting an SMT solver, the goal is encoded as an SMT-LIB [5] problem which is unsatisfiable iff the goal is valid. Sledgehammer also selects facts that it thinks will be relevant for solving the goal and includes encodings of them as well. The problem is given to the solver which reports back to Sledgehammer whether it was able to prove the goal [9]. Proving the goal externally, however, is not enough since Isabelle is a *skeptical* proof assistant, in the sense that it does not trust external solvers. Thus, a proof of the goal must somehow be constructed and checked inside Isabelle.

Finding such a proof internally can be challenging. One useful technique is to query the external solver for an *unsat core*, i.e., a subset of the facts it was given that are sufficient to prove the goal valid. Sometimes, this information is enough for Isabelle to search for an internal proof on its own. However, this process can be greatly improved, if, instead of just communicating the result and the core back to Sledgehammer, the solver also communicates a fine-grained proof. Then, with the appropriate proof reconstruction machinery, each step in the proof can be reconstructed as one or more steps using Isabelle’s internal inference engine. As mentioned in Section 1.1, Sledgehammer can do this for proofs from the veriT and z3 solvers, though the former supports only a limited set of theories, and the latter produces only coarse-grained proofs.

Still, this means that Isabelle already has an integration with solvers supporting the SMT-LIB standard and is able to translate to and from SMT-LIB and internal terms. We build on this integration and extend it. Notice that such an integration requires each SMT-LIB operator to be matched with a term in Isabelle with the same semantics. Isabelle has built-in operators that match well with those in the uninterpreted function and arithmetic SMT theories, and both

formalisms support quantifiers [18]. However, Isabelle only has partial support for bit-vector operators. A more complete development of bit-vectors in Isabelle is described by Böhme et al. [11], but unfortunately, parts of their work (including parsing bit-vector proofs) never made it into Isabelle and now appear to be lost. As we describe below, part of our effort includes improving support for SMT theories in Isabelle, including bit-vectors and strings.

2.4 Approximate Sorts

RARE rules are meant to be easy and effortless to write. This is not the case when users have to specify sort information that is either inferable from the context or too restrictive. As an example of the latter, consider any rewrite rules involving bit-vector sorts. The SMT-LIB standard provides bit-vectors sorts that are parameterized by their size, or *bit-width*. However, to keep sort checking simple, it requires all bit-widths in SMT-LIB scripts to be concrete as, for instance, in `(_ BitVec 8)`. A similar argument applies to polymorphic sorts because, although SMT-LIB allows the definition of theories with such sorts (such as, for instance, array, set, and sequence sorts), it restricts scripts to monomorphic instantiations of polymorphic sorts — e.g., `(Set Int)`.

Unfortunately, these restrictions are too strong for RARE. They make it impossible, for example, to write any rewrite rule involving bit-vector terms that is naturally parametric in the bit-width of those terms, or any rule involving terms with a polymorphic sort. The ideal solution would then be to introduce dependent types (or sorts, to maintain the SMT-LIB terminology) in RARE, allowing both value and type parameters in sorts — e.g., `(_ BitVec n)` with *n* an integer variable, and `(Array A B)` with *A* and *B* type variables. However, this would make it difficult for SMT solvers, CVC5 included, to process RARE rules since, effectively, they only support non-dependent, monomorphic sorts.

RARE's compromise solution is to add instead *approximate sorts* to the sort system, following an approach analogous to gradual typing in programming languages [33], a hybrid type-checking discipline where some program types are checked statically and others are checked dynamically. In our case, where there is no notion of dynamic checking, we have instead two sort-checking phases in the SMT solver for RARE rules: (i) as the rules are read by the solver, when sort checking is done with respect to the declared approximate sorts, and (ii) during proof elaboration, when the approximate sorts in the RARE rules are matched against the exact sorts in the proof steps that correspond to those rules.

Approximate sorts are obtained by extending the sort system of SMT-LIB with a distinguished unknown value and a distinguished unknown sort, both denoted by `?`, that can be used in place of a value or parameter in a sort. This allows the construction of approximate sorts such as `(_ BitVec ?)`, `(Set ?)`, and `(Array ? ?)` (abbreviated as `?BitVec`, `?Set`, and `?Array`), while still allowing precise sorts such as `(_ BitVec 1)`, `(Set Real)`, and `(Array Int Real)`. Approximate sorts can be used to approximate dependently-sorted/polymorphic rewrite rules, as we see in the next section.


```

⟨rule⟩      ::= ( define-rule ⟨symbol⟩ ( ⟨par⟩* ) [(⟨defs⟩) ⟨expr⟩ ⟨expr⟩ )
               | ( define-rule* ⟨symbol⟩ ( ⟨par⟩* ) [(⟨defs⟩) ⟨expr⟩ ⟨expr⟩ [(⟨expr⟩) ] )
               | ( define-cond-rule ⟨symbol⟩ ( ⟨par⟩* ) [(⟨defs⟩) ⟨expr⟩ ⟨expr⟩ ⟨expr⟩ ) )
⟨par⟩       ::= ⟨symbol⟩ ⟨sort⟩ [:list]
⟨sort⟩      ::= ⟨symbol⟩ | ? | ?⟨symbol⟩ | ( ⟨symbol⟩ ⟨numeral⟩+ )
⟨expr⟩      ::= ⟨const⟩ | ⟨id⟩ | ( ⟨id⟩ ⟨expr⟩+ )
⟨id⟩        ::= ⟨symbol⟩ | ( ⟨symbol⟩ ⟨numeral⟩+ )
⟨binding⟩   ::= ( ⟨symbol⟩ ⟨expr⟩ )
⟨defs⟩      ::= ( def ⟨binding⟩+ )

```

Fig. 1: Overview of the grammar of RARE.

An additional advantage of this approach is that, by relieving the RARE user from the burden of specifying the precise sort of variables in rewrite rules, it makes them both easier to write and less error-prone. At the same time, the loss of precision introduced by approximate sorts is not a serious hindrance in practice: both the SMT solver, which relies on RARE rules for proof elaboration, and IsaRARE, which uses them during proof reconstruction, are able to infer the exact sort represented by an approximate one thanks to their knowledge of the (exact) sort of the constant and function symbols in the supported SMT theories. Subsection 4.3 explains how IsaRARE recovers exact sorts by type inference fully automatically during the translation to Isabelle.

3 The RARE Language

The RARE language⁶ was introduced by Nötzli et al. [28]. As part of this work, we have extended the language to be able to represent more rewrite rules. We present the full updated language here and summarize the differences with [28] at the end of the section.

A RARE file contains a list of rules whose syntax is defined by the grammar in Figure 1. Expressions use SMT-LIB syntax with a few exceptions. These include the use of approximate sorts for parameterized sorts (e.g., arrays and bit-vectors) and the addition of a few extra operators (e.g., `bvsize`, described below). RARE uses SMT-LIB 3 syntax [6], which is very close to SMT-LIB 2 and mostly differs from its predecessor in that it uses higher-order functions for indexed operators.

We say that an expression e *matches* a match expression m if there is some *matching substitution* σ that replaces each variable in m by a term of the same sort to obtain e (i.e., $m\sigma$ is syntactically identical to e). For example, the expression (or (bvugt x_1 x_2) (= x_2 x_3)), with variables x_1 , x_2 , x_3 , all of sort `?BitVec`,

⁶ RARE comes from Rewrites, Automatically REconstructed.

matches $(\text{or } (\text{bvugt } a \ b) \ (= \ b \ a))$ but not $(\text{or } (\text{bvugt } a \ b) \ (= \ c \ a))$, with a , b , and c bit-vector constant symbols of the same bit-width.

RARE Rules A RARE rewrite rule is defined with the `define-rule` command which starts with a parameter list containing variables with their sorts. These variables are used for matching as explained below. After an optional *definition list* (see below), there follow two expressions that form the main body of the rule: the *match* expression and the *target* expression. The semantics of a rule with match expression m and target expression t is that any expression e matching m under some sort-preserving matching substitution σ can be replaced by $t\sigma$. With approximate sorts, the sort preservation requirement is relaxed as follows. In RARE, for any sort constructor S of arity $n > 0$, there is a corresponding approximate sort $(S \ ? \ \dots \ ?)$ with n occurrences of `?` which is always abbreviated as `?S`. A variable x with sort `?S` (e.g., `?BitVec`) in a match expression matches all expressions whose sort is constructed with S (e.g., $(\text{BitVec } 1)$, $(\text{BitVec } 2)$, and so on). Variables with sort `?` match expressions of any sort.

An optional *definition list* may appear in a RARE rule immediately after the parameter list. It starts with the keyword `def` and provides a list of local variables and their definitions, allowing the rewrite rule to be expressed more succinctly. A rule with a definition list is equivalent to the same rule without it, where each variable in the definition list has been replaced by its corresponding expression in the body of the rule. For a rule to be well-formed, all variables in the match and target expressions must appear either in the parameter list or the definition list. Similarly, each variable in the parameter list must appear in the match expression (while this second requirement could be relaxed, it is useful for catching mistakes). Consider the following example.

```
(define-rule bv-sign-extend-eliminate ((x ?BitVec) (n Int))
  (def (s (bvsized x)))
  (sign_extend n x) (concat (repeat n (extract (- s 1) (- s 1) x)) x))
```

In this rule, there are two parameters, x and n . The sort annotation `?BitVec` indicates that x is a bit-vector without specifying its bit-width. The latter is stored in the local variable s using the `bvsized` operator. The rule says that a $(\text{sign_extend } n \ x)$ expression can be replaced by repeating n times the most significant bit of x and then prepending this to x .

The `define-cond-rule` command is similar to `define-rule` except that it has an additional expression, the *condition*, immediately after the parameter and definition lists. This restricts the rule's applicability to cases where the condition can be proven equivalent to true under the matching substitution. In the example below, the condition $(> \ n \ 1)$ can be verified by evaluation since in SMT-LIB, the first argument of `repeat` must be a numeral.

```
(define-cond-rule bv-repeat-eliminate-1 ((x ?BitVec) (n Int))
  (> n 1) (repeat n x) (concat x (repeat (- n 1) x)))
```

Note that the rule does not apply to terms like $(\text{repeat } 1 \ t)$ or $(\text{repeat } 0 \ t)$.

Fixed-point Rules The `define-rule*` command defines rules that should be applied repeatedly, to completion. This is useful, for instance, in writing rules that iterate over the arguments of n -ary operators. Its basic form, with a body containing just a match and target expression, defines a rule that, whenever it is applied, must be applied again on the resulting term until it no longer applies.

The user can optionally supply a *context* to control the iteration. This is a third expression that must contain an underscore. The semantics is that the match expression rewrites to the context expression, with the underscore replaced by the target expression. Then the rule is applied again to the target expression only. In the example below, the `:list` modifier is used to represent an arbitrary number of arguments, including zero, of the same type.

```
(define-rule* bv-neg-add ((x ?BitVec) (y ?BitVec) (zs ?BitVec :list))
  (bvneg (bvadd x y zs)) (bvneg (bvadd y zs)) (bvadd (bvneg x) _))
```

This rule rewrites a term `(bvneg (bvadd s t ...))` to the term `(bvadd (bvneg s) r)` where r is the result of recursively applying the rule to `(bvneg (bvadd t ...))`.

Changes to RARE Here, we briefly mention the changes to RARE with respect to [28]. First, we have support for a richer class of approximate sorts, including approximate bit-vector and array sorts. Also, we replaced the `let` construct by the new `def` construct. The definition list is more powerful as it applies to the entire rest of the body (whereas `let` was local to a single expression).

Additionally, to aid with bit-vector rewrite rules, we added several operators: `bvsize`, which returns the width of an expression of sort `?BitVec`; `bv`, which takes a integer n and natural w , and returns a bit-vector of width w and value $n \bmod 2^w$; `int.log2` which returns the integer (base 2) logarithm of an integer, and `int.islog2`, which returns true iff its integer argument is a power of 2.

We also removed the `:const` modifier, which was used previously to indicate that a particular expression had to be a constant value. We found that this adds complexity and is usually unnecessary. For rules that actually manipulate specific constant values, we can specify those values explicitly, e.g., by using the `bv` operator above.

4 IsaRARE: from RARE Rewrites to Isabelle Lemmas

In this section, we introduce IsaRARE, a plugin for Isabelle that automatically translates a RARE rule into an Isabelle lemma stating the correctness of the rule. Being able to generate such lemmas automatically is highly desirable, as RARE rules may be added and/or changed frequently for a given solver, or differ significantly between solvers, and manually translating RARE rules into lemmas is time-consuming and error-prone. IsaRARE can also suggest a proof sketch which is sometimes sufficient to prove the lemma. If this automatic proof fails, the user must provide the proof or determine that the lemma does not hold. In the latter case, Isabelle’s counterexample-finder Nitpick [10] can be helpful.

```

(define-cond-rule str-len-replace-inv ((t String) (s String) (r String))
  (= (str.len s) (str.len r))
  (str.len (str.replace t s r)) (str.len t))

lemma str_len_replace_inv:
  fixes t::string and s::string and r::string
  shows "smtlib_str_len s = smtlib_str_len r  $\longrightarrow$ 
    smtlib_str_len (smtlib_str_replace t s r) = smtlib_str_len t"

```

Fig. 2: RARE rule and corresponding lemma.

Figure 2 shows an example of a RARE rule (which simplifies the length of the result of a string replacement) and the Isabelle lemma generated from it by IsaRARE. Roughly speaking, a rule with parameters x_1, \dots, x_m , definition list $((y_1 d_1) \cdots (y_n d_n))$, condition c , match expression s , and target expression t is converted by IsaRARE into a lemma of the form $\forall x_1, \dots, x_m. (c \Rightarrow s = t)\sigma$ where σ is the substitution $\{y_1 \mapsto d_1, \dots, y_n \mapsto d_n\}$. Type inference in Isabelle is used to suitably instantiate the `?` wildcards in any approximate sorts in the rules.

Next we discuss the main challenges we encountered while implementing the translation from RARE to Isabelle.

4.1 Adding New Theories

Since IsaRARE uses Isabelle’s SMT-LIB parser, it was necessary to extend it to handle SMT theories not previously supported and, in case there was no corresponding Isabelle theory, to define new types, definitions and theorems corresponding to the SMT-LIB theory. For sets and arrays, Isabelle already provides the required data structures (`Set.set` and `Map.map` respectively) and definitions (e.g., `union`, and `store`). Translation from the SMT operators and types is thus straightforward, requiring only simple extensions to the parser.

The SMT-LIB parser also had to be extended for the operators and sorts of the SMT-LIB theory of strings. String terms are represented with Isabelle’s `HOL.string`, and regular expressions are represented as sets of strings. We developed a new theory with auxiliary definitions and theorems meant to facilitate the proving of lemmas generated by IsaRARE. Since strings are defined as lists of characters, we were able to reuse many list operators for our definitions. For example, string concatenation is defined as concatenation of lists.

As mentioned, bit-vectors are encoded in Isabelle using the `word` type, which represents integers modulo 2^n , where n is a type parameter (see Subsection 4.3). Isabelle has support for reasoning about this type, but we still had to provide a number of extensions. For example, to translate bit-vector rewrite rules, we had to extend Isabelle’s SMT-LIB parser significantly. We added support for all of the standard SMT-LIB operators, as well as some additional operators that CVC5

<pre> (define-rule bv-extract-extract ((x ?BitVec) (i Int) (j Int) (k Int) (l Int))) (extract l k (extract j i x))) (extract (+ i l) (+ i k) x) </pre>	<pre> t0 = (extract j i x) ∧ size t0 = j + 1 - i ∧ t1 = (extract l k t0) ∧ size t1 = l + 1 - k ∧ t2 = (extract (i+1) (i+k) x) ∧ size t2 = (i+1) + 1 - (i+k) ∧ j < size x ∧ 0 ≤ i ∧ i ≤ j ∧ l < size t0 ∧ 0 ≤ k ∧ k ≤ l ∧ (i+1) < size x ∧ 0 ≤ (i+k) ∧ (i+k) ≤ (i+1) </pre>
--	---

(a) A RARE rule

(b) Additional Assumptions

Fig. 3: Implicit Assumption Generation

supports, such as `bvaddo` (which checks for overflow from unsigned addition). It was also necessary to add several new definitions and basic theorems to Isabelle, for example for reasoning about the `extract` operator.

4.2 Mismatch between Isabelle and SMT-LIB operators

An important challenge for the translation concerns the mismatch between SMT-LIB operators and Isabelle functions. One of the main difficulties concerns *implicit* assumptions. As an example, consider the bit-vector `extract` operator. The term `(extract i j t)` denotes the sub-vector of bit-vector t from index i through index j , where i is the more significant index. SMT-LIB specifies that the second index j must be at most i , and both indices must be in the range $[0, n)$, where n is the bit-width of t — making the result a bit-vector of width $i + 1 - j$. These assumptions are necessary to correctly capture the semantics of SMT-LIB’s `extract` since the `extract` operator in Isabelle is more permissive.

There are several ways to address this issue. First, we could make the implicit assumptions explicit in the RARE rules. However, this would be tedious and error-prone and would greatly clutter the RARE rules. It is also superfluous to always manually add them since the constraints are inherent in the SMT-LIB semantics. A second option is to write custom definitions for SMT-LIB operators in Isabelle that exactly match the SMT-LIB semantics (i.e., are undefined if the implicit assumptions do not hold). The main disadvantage of this approach is that it complicates proving the translated RARE rules, as those proofs cannot directly use any existing Isabelle lemmas that use the standard definitions. It also works against one of our long-term goals, which is to be able to use proof reconstruction to provide proofs for Isabelle conjectures, conjectures which will naturally use the existing Isabelle operators.

The last option, which we adopted, is to automatically add the implicit assumptions during the translation of RARE rules to Isabelle lemmas. This does make the lemmas a bit more complicated, but it is the minimal complexity needed to bridge the semantic gap between the two `extract` operators. And, we can be confident that these implicit assumptions will easily be discharged when using the lemmas for proof reconstruction, since SMT proofs only use operators

in ways that are consistent with SMT-LIB semantics (unless there is a bug, in which case proof reconstruction *should* fail). Figure 3 shows an example of a RARE rule with three applications of the `extract` operator, together with the assumptions added by IsaRARE.

In a few cases, we had to fall back on the custom definition approach. For example, we had to do this for the bit-vector `concat` operator for bit-vector concatenation. To see why, note that the SMT-LIB operator can take two or more arguments (abbreviating nested binary applications), each with arbitrary bit-width. Recall that the `:list` annotation in RARE can be used to specify a variable number of arguments. There is no way to even state lemmas corresponding to rewrite rules involving concatenations of a variable number of arguments in Isabelle using its built-in binary concatenation operator. For this case, we thus define a custom concatenation operator that matches the SMT-LIB semantics. The implicit assumption that the bit-width of the result is the sum of the bit-widths of the arguments is embedded in the custom definition. Using the new definition, we can translate the problematic rules into Isabelle lemmas. As expected, proving these lemmas requires extra work. Specifically, it requires formulating and proving bridging theorems between Isabelle’s built-in concatenation operator and the new one we defined.

4.3 Supporting Approximate Sorts

With the addition of approximate sorts to RARE, we had to extend Isabelle’s SMT-LIB translator to support them. We observe that Isabelle/HOL is not based on a dependently-typed logic. However, it supports an encoding of sorts depending on integer values into polymorphic types with parameters that range over types expressing ordinals. In particular, bit-vectors of width w are represented by the type `(n word)` of integers modulo 2^w ; for instance, `3::(8 word)` represents an integer with value 3 modulo 2^8 . In fact, thanks to polymorphism, it is possible for the bit-width to be a type variable (e.g., `3::('a::len word)`). Note that this is more precise than allowing the bit-width in the type to be completely unknown, as in approximate sorts: with type parameters one can state, for instance, that two terms of unknown bit-width have the same width, whereas two terms both of sort `?BitVec` may have different bit-widths.

Conveniently then, all the approximate sorts in RARE correspond to polymorphic types in Isabelle. For instance, `?BitVec` corresponds to `'a word` and `?Array` corresponds to `('a, 'b) map` where `'a` and `'b` are type variables. During parsing, each occurrence of a approximate sort is converted into an instance of the corresponding polymorphic type obtained by instantiating each sort variable with a fresh dummy type. For some bit-vector operators, the output sort is dependent on the input sorts (e.g., `extract` and `concat` as mentioned above). For applications of such operators, we also use a dummy type for the bit-width of each argument for which the width is not known. Once translation is done, we use Isabelle’s type inference algorithm to concretize each dummy type to a monomorphic one. For example, during translation of the rule `bv-ugt-eliminate` below, the variables `x` and `y` would both be assigned dummy types.

```
(define-rule bv-ugt-eliminate ((x ?BitVec) (y ?BitVec))
  (bvugt x y)    (bvult y x)
)
```

However, `bvugt` requires that both of its arguments be bit-vectors of the same width in SMT-LIB. This restriction is either already present in the definition in Isabelle that we map an operator to, or added during parsing as an implicit assumption, as we describe in Section 4.2. The type inference algorithm then computes the most general type for `x` and `y` that satisfies all assumptions. In this case, it correctly infers that they are bit-vectors of arbitrary but equal bit-width.

4.4 List Parameters

As mentioned earlier, SMT-LIB supports multi-arity syntax for certain binary operators, and RARE supports a variable number of arguments via the `:list` annotation. In contrast, in Isabelle all operators are fixed-arity. To facilitate the translation in these cases we added a new datasort, `'a rare_ListVar`, with a single constructor `ListVar :: 'a list \rightarrow 'a rare_ListVar` to encapsulate multiple arguments in a list. We also introduced two second-order operators, called `rare_list_left` and `rare_list_right`, to encode RARE left-associative and right-associative operators, respectively. As an example, a Boolean term of the form `(and $x_1 \dots x_n y z$)` is translated to the Isabelle term `(rare_list_right (\wedge) (ListVar [x1, ..., xn]) (y \wedge z))`. The `rare_list_left` and `rare_list_right` functionals fold the operator passed as first argument over the list stored in their second argument to obtain properly nested binary applications. For example, if $n = 2$, the Isabelle term above is translated to `(x1 \wedge (x2 \wedge (y \wedge z)))`.

For every multi-arity SMT-LIB operator, we prove that it can be built up from Isabelle's built-in binary version using `fold(r)` functions. For RARE rules with list parameters, these *transfer lemmas* become part of the correctness proof automatically generated by IsaRARE. When proving the corresponding lemma, we can take advantage of the many lemmas in Isabelle's libraries about fold functions without having to know the internals of the translation process.

If we have a RARE rule in which all arguments to an operator are lists, we must handle the special case when the lists are all empty. When the operator has an identity element, we return that. For example, applications of `and` to just empty lists are translated as standing for `true`. So far, we have only encountered one operator without an identity: bit-vector concatenation. Since neither SMT-LIB nor Isabelle support bit-vectors of bit-width 0, for that operator, we explicitly add an assumption ruling out the case where all lists are empty.

4.5 Writing Lemmas and their Proofs

To generate a lemma from a RARE rewrite rule, IsaRARE first introduces the parameters with their types using Isabelle's `fixes` construct. Next, it generates the statement of the lemma, the *goal*, which states that the implicit assumptions and conditions imply the equality of the match and target terms. The types of any

bit-vector constants are fully specified (via type ascription), because otherwise the lemma may be too general and not hold.

Lastly, IsaRARE adds an Isabelle proof of the lemma. For lemmas that do not contain lists, this is simply a call to the main automatic tactic `auto`. Otherwise, the list constructs are eliminated as explained above, and any transfer lemmas are applied to the resulting terms. This ensures that goals will not contain any IsaRARE list definitions. We then invoke induction for every list and use the `simp_all` tactic to attempt to solve and simplify the goals.

The proof is printed in *apply* style so that it can be easily modified and completed manually if Isabelle is unable to discharge all its sub-goals automatically.

4.6 Availability

IsaRARE currently supports the theories of uninterpreted functions, linear arithmetic, bit-vectors, arrays, strings, and sets. It is publicly available⁷ under the BSD 3-Clause license. We plan to submit IsaRARE to the Archive of Formal Proofs [20]. We have also been working with the Isabelle maintainers to have our extensions to Isabelle itself (e.g., to the SMT-LIB parser) included in the official Isabelle distribution. Many features were already included in the latest release. IsaRARE requires the Word_Lib library (which is also included in the Archive of Formal Proofs) if it is used on RARE rules containing bit-vector operators not present in Isabelle itself.

5 Evaluation and Experience

We used IsaRARE to help develop, translate, and verify new RARE rewrite rules. These rules were designed to address coarse-grained rewrite steps appearing in CVC5 proofs, i.e., steps that could not be elaborated into fine-grained steps using the existing RARE rules and the approach mentioned in Section 2.2. In this section, we report on this experience and also discuss challenges arising from particular rewrites and theories.

5.1 Impact of New Rewrites on CVC5 Proof Holes

Previous work developed 85 RARE rules for CVC5 [28]. For our evaluation, we ran CVC5 with these plus our 163 new rules, bringing the total number of RARE rules in the CVC5 database to 248. We evaluated the impact of the new rules on CVC5's ability to produce fine-grained proof steps by comparing the *success rate* of the elaboration (i.e., percentage of rewriting proof steps that are successfully elaborated into fine-grained steps) before and after the addition of the new rules. We ran CVC5 on 70,709 unsatisfiable benchmarks, as determined by CVC5 [2, Sec. 4], in the SMT-LIB logics containing quantifier-free problems with equality and uninterpreted functions, arrays, linear arithmetic, strings, and bit-vectors.

⁷ <https://github.com/cvc5/IsaRARE>

theory	rewrites		proven	autoproven
	old	new		
EUF	22	43	43	37
Arithmetic	23	22	22	14
Sets	0	7	7	7
Arrays	0	4	4	4
Strings	40	57	57	37
Bit-vectors	0	115	84	62

Table 1: Rule and rule verification counts per theory

The results were generated with a cluster equipped with 16 x Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz, 62.79 GiB RAM machines, with one core per solver/benchmark pair, 1200s time limit, and 8gb memory limit.

For string benchmarks (the only set evaluated in [28]), the success rate went from 92% to 98%. Results on the logics with equality and uninterpreted functions, arrays, and linear arithmetic were similar. By far the most challenging theory, in terms of rewrite rules, is the bit-vector theory. Prior to our work, there were no RARE rules for this theory, so no bit-vector rewrite steps could be turned into fine-grained steps. With our 115 new RARE rules for bit-vectors, 92% of coarse-grained bit-vector rewrite steps are successfully elaborated into fine-grained steps. We see this as tremendous progress towards full fine-grained proofs for bit-vector problems.

5.2 Translating and Verifying Rewrites

In Table 1, we list the number of new rules in each theory, distinguishing between how many were there before (old) and the total including both the old rules and our new rules (new).⁸ We also show how many of the lemmas we have successfully proven and how many of these were done automatically, i.e., either by the proof suggested by IsaRARE or by a single call to Sledgehammer. The **proven** column shows that all non-bit-vector rules as well as most of the bit-vector rules have now been proven. The numbers in the last column show that most of the proofs were provided automatically by IsaRARE.

For the theory of strings, the number of lemmas automatically proven is not clear-cut. For other theories, libraries with useful background lemmas already existed, but for strings we had to add many new general-purpose lemmas ourselves and then decide whether these should count as background lemmas or as part of the proof effort for a rewrite rule. We were rather conservative in that decision, i.e., we did not count a lemma as automatically proved if it used a lemma whose classification as a background lemma was in doubt. Many of the

⁸ Consolidation in the set of arithmetic rules actually resulted in one fewer rule than existed previously.

translated string rewrites had to be proved manually because they required induction on string length, especially since many operators are defined inductively. However, we found that most of these manual proofs were fairly easy once an appropriate induction variable was selected.

There are no performance issues—IsaRARE translates most files in milliseconds. Even for our biggest RARE database, the one containing bit-vector rules, IsaRARE took only around 1-2 seconds on our machine.

5.3 Bugs Found in String Rules

We found several bugs in the existing RARE rules for strings by using Isabelle’s counterexample finder Nitpick [10] on the translated Isabelle lemmas. We diagnosed and fixed each of them, so that now they can all be verified.⁹ The bugs fall into three main categories.

Misinterpreted Semantics: The `str.substr` operator takes three arguments and returns the substring of the first argument, starting at the position given by the second argument, and continuing for the number of characters specified by the third argument. The following (corrected) rule simplifies a substring expression to the empty string whenever the third argument is 0 or negative.

```
(define-cond-rule str-substr-empty-range ((x String) (n Int) (m Int))
  (>= 0 m) (str.substr x n m) "")
```

However, the first version of the rule had the wrong condition: `(>= n m)` rather than `(>= 0 m)`. This is likely due to the rule’s author mistaking the third argument of `str.substr` for an absolute index instead of a relative offset.

Forgotten Condition: The corrected rule below says that, under some assumptions, the length of a substring term is equal to the offset (third) argument.

```
(define-cond-rule str-len-substr-in-range ((s String) (n Int) (m Int))
  (and (>= n 0) (>= m 0) (>= (str.len s) (+ n m)))
  (str.len (str.substr s n m)) m)
```

The earlier version of the rule did not include the condition `(>= m 0)`. This however, makes it unsound, because according to the semantics of `str.substr`, if the offset is negative, the result is just the empty string. This led to a counterexample with a negative value for `m`. Note that this condition is not automatically added by IsaRARE since `str.substr` is defined for negative offsets.

Misunderstanding the Rewrite: One rule was designed to closely mirror a piece of CVC5 code implementing a rewrite, but it failed to properly capture all cases. The code involved included several conditionals resulting in two different ways a term could be rewritten. The original rule only captured one of the two cases and even missed one of the conditions for the case it included. Since this rule was quite complex and was only incorrect for some corner cases, it would have been challenging to find this bug without our verification effort.

⁹ Fortunately, none of the bugs in rules corresponded to buggy code in CVC5 itself. However, CVC5 could have used those rules to construct incorrect proofs.

5.4 Bit-vector Rewrite Rules

Bit-vector theory solvers make extensive use of rewriting, employing large numbers of rewrite rules. In order to define RARE rules for CVC5’s bit-vector theory, we began by analyzing the CVC5 rewriting code, which implements a total of 99 rewrite methods. We then wrote RARE rules to try to capture the behavior of these methods. There are 5 methods that are too complex to be captured by RARE (or by any straightforward extension of it). For each of these, we instead added new hard-coded proof rules to the CVC5 proof rule database.¹⁰ These hard-coded proof rules are not included in Table 1, but they *are* used to help demonstrate the overall progress on SMT-LIB proofs (Section 5.1). The long-term plan for reconstruction of proofs using these rules is to write custom Isabelle tactics for reconstructing those proof steps.

Unlike with the string rules, where we applied IsaRARE to already-written rules, we used IsaRARE extensively to help debug the bit-vector rules as they were being written. We were able to quickly and easily find many kinds of mistakes this way. For example, rule authors mixed up `bvneg` (unary 2’s complement negation) and `bvnot` (bit-wise Boolean negation). In other cases, rules used inconsistent bit-widths. The type inference that IsaRARE performs is particularly helpful in such cases, as it is stricter than the CVC5 RARE parser.

Many of the bit-vector rules can be proved automatically, but others must be proved manually and are quite challenging, especially those involving signed arithmetic or division. Despite this, as shown in Table 1, the process of manually proving the full set of bit-vector lemmas is largely complete. This is important for our long-term goal of reconstructing SMT proofs in Isabelle.

6 Conclusion

We presented IsaRARE, a tool providing an automatic pipeline for verifying rewrite rules. We showed the effectiveness of our approach by proving the correctness of a large number of rewrite rules used in CVC5 proofs. Our experiments show that many lemmas can be proved with minimal user interaction.

This work is also part of a long-term project that aims to further automate proof search in Isabelle. The goal is to be able to reconstruct any CVC5 proof in Isabelle’s internal inference engine. This, of course, also includes reconstructing rewrite steps. The lemmas IsaRARE generates are directly applicable to this effort. We plan to provide a detailed description and evaluation of this larger effort in future work.

Data Availability Statement The datasets generated and analyzed during the current study are available in the Zenodo repository: <https://zenodo.org/records/10048664> [24].

¹⁰ This is analogous to the handling of polynomial normalization in [28].

References

1. Backes, J., Bolognani, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9. IEEE (2018)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., et al.: cvc5: a versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
3. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning* **64**(3), 485–510 (2020)
4. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) *Automated Reasoning*. pp. 15–35. Springer International Publishing, Cham (2022)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
6. Barrett, C., Fontaine, P., Tinelli, C.: SMT-LIB Version 3.0 - Preliminary Proposal (2021), <https://smtlib.cs.uiowa.edu/version3.shtml>
7. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) *All about Proofs, Proofs for All, Mathematical Logic and Foundations*, vol. 55, pp. 23–44. College Publications, London, UK (Jan 2015)
8. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications*, vol. 336, pp. 1267–1329. IOS Press (2021)
9. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) *Automated Deduction – CADE-23*. pp. 116–130. Springer Berlin Heidelberg (2011)
10. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) *Interactive Theorem Proving*. pp. 131–146. Springer Berlin Heidelberg (2010)
11. Böhme, S., Fox, A.C., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: *International Conference on Certified Programs and Proofs*. pp. 183–198. Springer (2011)
12. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) *Interactive Theorem Proving*. pp. 179–194. Springer, Berlin, Heidelberg (2010)
13. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) *Automated Deduction – CADE-22*. pp. 151–156. Springer (2009)
14. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) *Model Checking Software*. pp. 248–254. Springer (2012)
15. Cook, B.: Formal reasoning about the security of Amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) *Computer Aided Verification*. pp. 38–47. Springer (2018)

16. Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
17. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) *NASA Formal Methods. Lecture Notes in Computer Science*, vol. 11460, pp. 148–165. Springer (2019)
18. Fleury, M., Schurr, H.J.: Reconstructing veriT proofs in isabelle/HOL. *Electronic Proceedings in Theoretical Computer Science* **301**, 36–50 (2019)
19. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) *International Workshop on Satisfiability Modulo Theories (SMT). CEUR Workshop Proceedings*, vol. 3185, pp. 54–70. CEUR-WS.org (2022)
20. Jaskelioff, M., Merz, S.: Proving the correctness of disk paxos. *Archive of Formal Proofs* (June 2005), <https://isa-afp.org/entries/DiskPaxos.html>, Formal proof development
21. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: a certified string solver. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. pp. 210–224. Association for Computing Machinery (2022)
22. Katz, G., Barrett, C., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for DPLL (T)-based SMT solvers. In: Piskac, R., Talupur, M. (eds.) *2016 Formal Methods in Computer-Aided Design (FMCAD)*. pp. 93–100. IEEE (2016)
23. 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. Association for Computing Machinery (2009)
24. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Noetzli, A., Barrett, C., Tinelli, C.: IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL (Oct 2023), <https://doi.org/10.5281/zenodo.10048664>
25. de Moura, L., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops. CEUR Workshop Proceedings*, vol. 418. CEUR-WS.org (2008)
26. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer (2008)
27. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
28. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) *2022 Formal Methods in Computer-Aided Design (FMCAD)*. p. 65 (2022)
29. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-specific proof steps witnessing correctness of SMT executions. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. pp. 541–546. IEEE (2021)
30. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. *Formal Aspects of Computing* **31**(6), 675–698 (2019)
31. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) *Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science*, vol. 12699, pp. 450–467. Springer (2021)
32. Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). *Electronic Proceedings in Theoretical Computer Science* **336**, 49–54 (2021)

33. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: ACM (ed.) Proceedings of Scheme and Functional Programming Workshop (2006)

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.





Automate where Automation Fails: Proof Strategies for Frama-C/WP

Loïc Correnson¹, Allan Blanchard¹, Adel Djoudi²,
and Nikolai Kosmatov³

¹ Université Paris-Saclay, CEA, List, Palaiseau, France
{loic.correnson,allan.blanchard}@cea.fr

² Thales Digital Identity & Security, Meudon, France
adel.djoudi@thalesgroup.com

³ Thales Research & Technology, Palaiseau, France
nikolai.kosmatov@thalesgroup.com

Abstract. Modern deductive verification tools succeed in automatically proving the great majority of program annotations thanks in particular to constantly evolving SMT solvers they rely on. The remaining proof goals still require interactively created proof scripts. This tool demo paper presents a new solution for an automatic creation of proof scripts in Frama-C/WP, a popular deductive verifier for C programs. The verification engineer defines a proof strategy describing several initial proof steps, from which proof scripts are automatically generated and applied. Our experiments on a large real-life industrial project confirm that the new proof strategy engine strongly facilitates the verification process by automating the creation of proof scripts, thus increasing the potential of industrial applications of deductive verification on large code bases.

Keywords: deductive verification, proof automation, interactive proof scripts, proof strategies, Frama-C.

1 Introduction

Recent years have seen many successful applications of deductive verification [7, 8]. Modern deductive verifiers manage to automatically prove the greatest number of *proof goals*, also called *proof obligations*, or *verification conditions* (VCs). This is in particular due to powerful and constantly evolving SMT solvers they rely on. The remaining unproven goals typically require some form of interactive proof: either with a proof script indicating a few initial proof steps to make the goal more suitable for an automatic prover, or a fully interactive proof in a proof assistant like Coq. The need for an interactive proof remains an important obstacle to a wider application of deductive verification on large projects.

It can be illustrated by a recent proof [6] of real-life smart card code—a JavaCard Virtual Machine (JCVM)—that was performed by Thales for the highest EAL6–EAL7 levels of Common Criteria certification⁴ using Frama-C/WP [9],

⁴ The EAL7 certificate delivered by the French certification body ANSSI is available at https://cyber.gouv.fr/sites/default/files/document_type/Certificat-CC-2023_45fr_0.pdf.

a popular deductive verifier for C programs. Even if a very high level of automation is achieved in that project and less than 2% of proof goals require manually created proof scripts, a significant effort is still required for the remaining goals because hundreds of properties are concerned.

Moreover, proof scripts are sensitive to the versions of the deductive verifier, of the code and the specification. Thus, proof scripts not only need to be created once for a given version of the target code, its specification and the verifier, but often have to be recreated when the code or the specification are updated, or the verifier evolves (and hence the way to generate VCs is modified). Thus, the need for manually created proof scripts for the unproven goals is seen as an important obstacle to a better maintenance of the proved code in the industrial setting.

This tool demo paper presents a new mechanism⁵ for an automatic creation of proof scripts in Frama-C/WP. The verification engineer defines a proof strategy describing the alternative proof steps to be tried, from which proof scripts are automatically generated and applied. Our experiments on the JCVM verification project confirm that the new mechanism strongly facilitates the verification process, thus increasing the potential of industrial applications of deductive verification on large code bases.

The contributions of this work include a demonstration of the new mechanism for automating the creation of proof scripts in Frama-C/WP based on user-defined proof strategies, its illustration on simple examples and its evaluation on a real-life industrial project.

2 Deductive Verification with Frama-C/WP

Frama-C is an open-source, industrially mature, extensible framework for verifying C programs annotated with ACSL [2] specifications. The WP plug-in of Frama-C allows the user to prove that the C code respects the ACSL specifications using *deductive verification* [7, 8]. More precisely, WP implements an efficient variant of *weakest precondition calculus* [10], hence the name of the plug-in.

ACSL specifications, written inside special comments “/*@...*/”, basically consist of function contracts and code annotations. Function contracts include pre-conditions (**requires** clauses) and post-conditions (**ensures** clauses), containing pure logical formulas that shall be verified respectively before and after any call to a function. The **assigns** clause specifies the possible side effects of the function on global variables and pointers received in parameters. Code annotations (e.g. **assert** clauses) contain pure logical formulas attached to a particular program point that shall be verified at each execution path going through this program point. These clauses are illustrated by the program below:

```

1  /*@
2    requires 0 ≤ x ≤ y ;
3    ensures \result == (x + y) / 2;
4    assigns \nothing;
5  */
6  int middle(int x, int y)

```

⁵ publicly available on <https://git.frama-c.com/pub/frama-c/> as part of the current development version (and in the upcoming release planned for November 2023).


```

7 {
8   /*@ assert 0 ≤ y - x < MAX_INT; */
9   /*@ assert 0 ≤ x + (y-x) / 2 < MAX_INT; */
10  return x + (y - x) / 2;
11 }

```

Frama-C contains plug-ins that can generate assertions, and plug-ins that can prove assertions, or both. For instance, the RTE plug-in can generate code annotations that are sufficient for the program to never go into unspecified or undefined behaviors. The assertions in the previous example (Lines 8–9) show two of the five assertions generated by RTE on this code.

WP is able to *prove* code annotations written by the user or generated by other plug-ins. It works by using *deductive verification*: ACSL logic formula and C-code instructions are translated to some equivalent pure logic formulæ in a first-order logic language. Each generated formula is first simplified by a built-in solver named Qed [4] and then submitted to external provers, generally automated SMT solvers such as Alt-Ergo, Z3, CVC4 or CVC5. On the above program, WP can prove all ACSL annotations written by the user and generated by RTE:

```

1 $ frama-c -wp -wp-rte middle.c
2 [rte:annot] annotating function f
3 [wp] 8 goals scheduled
4 [wp] Proved goals:      8 / 8
5   Qed:                  2
6   Alt-Ergo 2.4.2:       6 (4ms-14ms)

```

In this example, RTE generated 5 annotations and WP generated 8 formulas for proving all resulting ACSL annotations, 2 of which being proved by Qed simplification, and the remaining 6 being proved by Alt-Ergo in few milliseconds.

3 Automated vs. Interactive Proofs

In most cases, ACSL annotations are automatically proven by Qed and SMT solvers. However, sometimes an automated proof might fail for a correct formula because deductive verification is not complete in general, and WP in particular.

In such a situation, WP offers different features to complete the proofs. First, the user might help SMT solvers by introducing intermediate code annotations, hence providing proof hints and intermediate proof results. Second, the user might enter the interactive proof mode with the Frama-C graphical interface, in which the user can apply so-called *tactics* to transform a proof goal into a conjunction of several, typically simpler ones, that WP can try to prove in turn. This process can be iterated, and all the applied tactics can be saved on disk in a *proof script* file that can be replayed later from the Frama-C command line.

After some efforts, the user can thus manage to achieve full automation in proof replay for a proof campaign: all proof goals are discharged automatically by SMT solvers, possibly thanks to proof hints provided as code annotations, and possibly after applying tactics from saved proof scripts.

WP offers a large variety of tactics. Common ones include splitting over a boolean expression; brute-forcing an integer expression within a given range (detailed below); unfolding predicate or function definitions; removing hypotheses; etc. Applying tactics is simple in spirit, although it raises complex issues in

practice. Consider for instance the *Range* tactic, which can be defined as follows, where φ is the current goal, e some expression and $a \leq b$ two integer constants:

$$\text{range}(\varphi, e, a, b) \equiv \bigwedge_{k \in a \dots b} (e = k \implies \varphi) \wedge (e < a \implies \varphi) \wedge (e > b \implies \varphi)$$

Applying it on goal φ consists in replacing⁶ φ by $\text{range}(\varphi, e, a, b)$. It requires to have at hand the expression e and the two constants a and b . Under the graphical user interface (GUI), those arguments are selected by the user from the goal. However, bookkeeping them in a proof script is not that simple, especially if we want the proof script to resist to minor changes in the code or the specifications. WP has dedicated features to achieve this choice but up to a certain extent.

In practice, managing proof scripts during the lifetime of large projects *is* an industrial issue. On the contrary, proof hints in the form of intermediate code annotations are quite robust. However, writing code annotations by hand is tedious. On the other hand, applying tactics to decompose goals is quite efficient, and it appears that, on a given application, many pending goals are solved by applying few tactics with very similar patterns. Those observations lead us to the design of *proof strategies*.

4 Definition of Proof Strategies

This section introduces the main principles and selected features of proof strategies through illustrative examples, which can be tested using the companion artifact [5]. We refer the reader to the WP manual [1] for a detailed description.

Proof strategies are user-defined specifications for combining automated solvers with pattern-driven tactics. A proof strategy consists of a list of alternatives to be tried *in sequence* on a proof goal until success. Elementary alternatives consist in trying one or several SMT solvers with a specified timeout, or applying a tactic on a goal. Lists of alternatives can be grouped and given a (strategy) name, that can be used as an elementary alternative as well. Then, specific proof strategies can be associated to specific proof goals, functions or lemmas. For instance, the user may associate proof strategy A to every code annotation *with* name P and proof strategy B to every code annotation *without* name Q, and finally proof strategy C to other code annotations.

Proof strategies and their association to proof goals are user-written as specific ACSL extensions defined and managed by the Frama-C/WP plug-in. An overview of these annotations is provided below:

```
strategy strategyname : alternative , ... , alternative ;
proof      strategyname : target , ... , target ;
```

The **strategy** clause introduces a new proof strategy *strategyname*, whereas the **proof** clause associates it to some property *targets*, i.e. individual goals

⁶ We have $\text{range}(\varphi, e, a, b) \implies \varphi$, which is sufficient for the tactic to be safely applied.

or sets of goals, using the same syntax as for **frama-c** command line, which simplifies users' learning curve. As introduced above, each *alternative* might consist of:

- `\provers(p,...,p,time)` which tries the specified provers in sequence with a specified timeout.
- `\tactic(id,param...)` tries to apply the specified tactic with the associated parameter(s).
- *strategyname* or `\default` tries the specified named strategy.

Parameters for applying tactics are the most expressive but also the most complex components of proof strategies. As briefly introduced in previous section, a tactic transforms a proof goal into one or several sub-goals that are sufficient to entail the initial goal. The difficult point with tactics is that they need *parameters* to be applied. For instance, the tactic **range** illustrated in previous section must be applied to an *expression* and a range of two integer constants. From the Frama-C GUI, proof engineers often pick those parameters from the goal itself, according to some patterns of interest and their experience. Our proof strategy language allows proof engineers to specify those patterns, and to build tactic parameters with required values accordingly.

A trade-off between robustness and precise definition of tactic applications is an important design objective. The proposed strategy language allows a significant flexibility in choosing precise (and less robust) or more general (and more robust) patterns. The latter include `'_'` for any expression, `'..'` for any number of arguments, `'A:_'` to introduce a variable to name a subexpression and to use it in a tactic parameter or a pattern to select, etc.

Consider lemma **dn3** in Fig. 1, not proved by Alt-Ergo. It can now be proved by associating to it the following strategy (we omit surrounding `/*@...*/`):

```

1 strategy RangeThenProver:                5 \param("inf",0),\param("sup",255),
2 \tactic ("Wp.range",                      6 \children(RangeThenProver) ),
3 \pattern(is_uint8(e)),                   7 \prover("alt-ergo",2);
4 \select(e),                             8 proof RangeThenProver: dn3;
```

The `"Wp.range"` name identifies the **range** tactic introduced above. This strategy looks for a variable *e* of type **unsigned char** (pattern `is_uint8(e)`, cf. Line 3) in the goal. If such a pattern is found in goal φ , the tactic **range**($\varphi, e, 0, 255$) is applied on φ (cf. Lines 2–5). Otherwise, the Alt-Ergo prover is applied for 2s (Line 7). The tactic specification language also offers directives to specify which strategies shall be applied on the resulting sub-goals. Line 6 above indicates that the strategy should be applied recursively. In this way, it enumerates first the values of *c*, then those of *d*. Indeed, the recursive application to all subgoals in this case is equivalent to selecting a first variable of type **unsigned char** and enumerating its values, then for each fixed value, doing so for a second variable of type **unsigned char** (and in this case, there are no more such variables). WP takes only ~ 1 s to automatically create the script and prove the lemma, while its manual creation would take several minutes.

Moreover, each sub-goal generated by applying a tactic has predefined names. For instance, tactic **range**(φ, e, a, b) generates a sub-goal named **"Lower a"** for

```

1 lemma dn3:
2   ∀ unsigned char c d;
3   (c & 0x8E) == 2 ∧
4   (c & 0x01) == 1 ∧
5   (d & 0x8F) == 0
6   ⇒ ((c+d) & 0x03) == 0x03;
7 lemma vhm_preserved{L1,L2}:
8   valid_heap_model{L1} ∧
9   mem_model_footprint_intact{L1,L2} ∧
10  \at(gNumObjs, L1) == \at(gNumObjs, L2) ∧
11  object_headers_intact{L1, L2}
12  ⇒ valid_heap_model{L2};

```

Fig. 1. Two ACSL lemmas not proved by automatic prover Alt-Ergo (with a 5 min. timeout).

```

1 strategy FastAltErgo: \prover("alt-ergo", 1); // run Alt-Ergo for 1s
2 strategy EagerAltErgo: \prover("alt-ergo", 10); // run Alt-Ergo for 10s
3 strategy UnfoldVhmThenProver: // Strategy with three steps:
4   FastAltErgo, // 1) fast prover attempt
5   \tactic("Wp.unfold", // 2) if unproved, unfold
6   \pattern(P_valid_heap_model((...))), // predicate valid_heap_model
7   \children(UnfoldVhmThenProver) ), // and apply itself recursively
8   EagerAltErgo; // 3) longer prover attempt
9 proof UnfoldVhmThenProver: vhm_preserved; // Associate strategy to goal

```

Fig. 2. Strategies to automatically create a proof script for lemma `vhm_preserved` of Fig. 1.

case $e < a$, "Upper b " for case $e > b$ and "Value k " for each case $e = k$ with $k \in a..b$. The user can then specify which strategy shall be used for each generated sub-goal. More detailed documentation can be found in the WP manual [1].

The second lemma in Fig. 1 comes from the example in [6] on the proof of the JCVM. It was not proved by the Alt-Ergo prover [3] (used in that work) and required a proof script. Basically, lemma `vhm_preserved` deduces predicate `valid_heap_model` at label (i.e. program point) L2 from the same predicate at label L1 (Lines 8, 12 in Fig. 1) if additional conditions are satisfied: the variables defining the memory state and the number of allocated objects do not change between labels L1 and L2 (Lines 9–10), and the headers of the allocated objects (indicating object owner, object size, etc.) do not change between labels⁷ L1 and L2 either (Line 11). Such lemmas are useful in large verification projects with lots of variables: by showing the preservation of values only for a few variables between two program points, this lemma allows the tool to deduce the predicate of interest at a new program point. The exact definition of predicates is not necessary to follow the present paper (and can be found in [6]).

With the presented extension of WP, the verification engineer can define a strategy `UnfoldVhmThenProver` (see Fig. 2) indicating which proof steps should be applied in order to achieve the proof. First, it calls the Alt-Ergo prover to check whether the goal can be proved with a short timeout (cf. Lines 4 and 1). If not, Lines 5–7 provide another alternative: to apply the *Unfold* tactic to unfold the definition of predicate `valid_heap_model` (in any part of the goal and with any number of arguments). Line 7 indicates that after a successful unfolding, the same strategy should be applied iteratively on the resulting sub-

⁷ Labels L1 and L2 can be C labels or predefined ACSL labels [2]. While labels are not directly preserved in the resulting VCs, the variables at those labels typically have different names, so it is still possible to match the corresponding values.

goals (children). Finally, Line 8 indicates that if the unfolding alternative cannot be applied anymore, a longer prover attempt is tried (cf. Line 2). This strategy allows WP to prove the target lemma in ~ 2 s.

5 Industrial Evaluation and Conclusion

We have applied the presented extension of Frama-C/WP to the proof of the real-life JCVM code⁸ (with 8,000+ lines of C and 30,000+ lines of ACSL) at Thales. The complete proof for 85,000 goals using Alt-Ergo with a 250s timeout requires 800+ proof scripts. The new tool saves a very significant effort: after a manual creation of strategies (~ 2 days), *WP automatically produces more than 50% of the required scripts, whose manual creation would take ~ 1 person-month*. This effort is estimated by the authors based on the experience of manual proof script creation in the industrial context over four years. In this experiment, the strategies are created by the same verification engineers who have previously created proof scripts. The same strategy is often able to successfully prove several dozens of proof goals, which confirms the reusability of strategies for multiple goals.

We summarize our experiment as a two-step workflow. First, the verification engineer creates proof strategies. Frequently used tactics (*Unfold*, *Split*, etc.) may be used as an initial guess with a large timeout in order to maximize proof automation. If some goals are still not proved, the engineer uses their experience to propose new ones, tuned to failed goals. The generated scripts are then saved for a proof replay session. Second, the engineer optimizes the strategies, e.g. by optimizing the script generation or replay time. The creation of strategies requires similar skills as for the creation of proof scripts.

We believe that an even greater number of proof scripts can be generated from strategies, which will strongly facilitate industrial verification. Future steps include identification and implementation of further strategy features, and their rigorous evaluation on various industrial projects. A detailed analysis of the reasons why some goals remained unproven in our experiment on the JCVM code will provide a better understanding of the nature of those goals and the required additional strategies. Finally, an evaluation of the usability of strategies by various categories of users (e.g. verification engineers who are not familiar with the target project or with proof scripts in Frama-C) is another future work perspective.

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018). We thank Nathan Koskas de Diego and Virgile Prevosto for many fruitful discussions and preliminary investigations that encouraged this work, as well as the anonymous referees for helpful comments.

⁸ Being highly security-critical, this code cannot be shared or included in an artifact.

References

1. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual (2023), <https://frama-c.com/download/frama-c-wp-manual.pdf>
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2021), <https://www.frama-c.com/download/acsl.pdf>
3. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: International Workshop on Satisfiability Modulo Theories(SMT 2018). <https://hal.inria.fr/hal-01960203>
4. Correnson, L.: Qed. Computing what remains to be proved. In: NASA Formal Methods Symp. (NFM 2014). LNCS, vol. 8430, pp. 215–229. Springer (2014)
5. Correnson, L., Blanchard, A., Djoudi, A., Kosmatov, N.: Automate where automation fails: Proof strategies for Frama-C/WP. Companion artifact for the paper submitted to TACAS 2024. (Nov 2023), <https://doi.org/10.5281/zenodo.10047833>
6. Djoudi, A., Hana, M., Kosmatov, N.: Formal verification of a JavaCard virtual machine with Frama-C. In: the 24th Int. Symp. on Formal Methods (FM 2021). vol. 13047, pp. 427–444. Springer (2021)
7. Filliâtre, J.: Deductive Software Verification. International Journal on Software Tools for Technology Transfer 13(5), 397–403 (2011)
8. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019)
9. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. pp. 1–37 (2015)
10. Leino, K.R.M.: Efficient Weakest Preconditions. Information Processing Letters 93(6), 281–288 (2005)

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

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





VeSCMul: Verified Implementation of S-C-Rewriting for Multiplier Verification

Mertcan Temel^(✉) 

Intel Corporation, Austin, TX, USA
mert.temel@intel.com

Abstract. Formal verification of multipliers is difficult. This paper presents a custom tool, VeSCMul, designed to address this problem. VeSCMul can be effectively applied to a wide range of hardware verification challenges, including multipliers with saturation, flags, shifting, truncation, accumulation, dot product, and even floating-point multiplication. The tool is highly automated with a user-friendly interface, and it is very efficient; for instance, verification for designs with 64-bit operands can finish in seconds. Notably, VeSCMul has been successfully utilized for both commercial designs and publicly available benchmarks. Regarding the reliability of its results, VeSCMul itself is fully verified, instilling confidence in its users for soundness. It also has the option to be used with a SAT solver for completeness and counterexample generation. Readers of this paper will gain insights into the capabilities and limitations of VeSCMul, as well as how to employ it for the verification of their own designs.

Keywords: Multipliers · Hardware Verification · Formal Methods

1 Introduction

Integer multipliers are crucial components in processing units. Ensuring their correctness through formal verification is essential; however, historically, verifying them has proven to be challenging [4,6,15,18]. Automated methods like SAT solving, BDDs, and computer algebra systems have either failed to scale or demonstrated limited applicability in this context [2,8,12,16,25]. On the other hand, the S-C-Rewriting method has been shown to be very efficient in formally verifying a large variety of RTL designs [21,24,25,26].

S-C-Rewriting and auxiliary programs are packaged into the VeSCMul tool (pronounced “vesk-muhl”). VeSCMul is designed to be user-friendly and comprehensive for sound, fast, and automatic verification of multiplier-centric RTL designs. It has an improved user interface tailored for non-experts, simplifying tool usage. VeSCMul has also introduced the support for fully automatic verification with its new adder detection program. VeSCMul has undergone extensive testing on thousands of public benchmarks as well as proprietary industrial designs at Intel Corporation. Its open-source and free-license status enables others to use this tool for similar verification tasks.

This paper presents VeSCMul, and it is outlined as follows. Sec. 2 walks through a demo for VeSCMul, showing the user-interface. Sec. 3 gives an overview of the tool flow. Sec. 4 lists some of the noteworthy features. Sec. 5 delivers experimental results on both public and proprietary designs. Sec. 6 discusses related work and concludes the paper.

2 Installation and a Demo

VeSCMul is implemented in the ACL2 theorem prover and programming language [10], and it is fully verified. VeSCMul is open-source with the MIT license, included as a Community Book in the ACL2 distribution on Github, which can be found at <https://github.com/acl2/acl2> under books/projects/vescmul. Installing ACL2 and building the books will bring along VeSCMul.

A comprehensive and up-to-date documentation for VeSCMul is available as part of ACL2’s manual, accessible at <http://acl2.org/manual>. This documentation is extensive, covering thousands of topics from ACL2 sources and Community Books. Throughout this paper, various documentation topics are referenced using the notation “:doc <topic>”.

Once ACL2 is installed and books are built, users can run a VeSCMul demo by running the events from Listing 1.1 within an ACL2 interactive session.

Listing 1.1: Simple demo running VeSCMul on a signed 64x64-bit multiplier with Booth radix-4 encoding, Dadda tree, and Han-Carlson adder.

```
(include-book "projects/vescmul/top" :dir :system)

(vescmul-parse
 :name      my-multiplier-example
 :file      "DT_SB4_HC_64_64_multgen.sv"
 :topmodule "DT_SB4_HC_64_64")

(vescmul-verify
 :name      my-multiplier-example
 :concl      (equal RESULT
                    (loghead 128 (* (logext 64 IN1)
                                     (logext 64 IN2)))))
```

The first event (`include-book`) loads VeSCMul and required libraries, which takes about a minute. Alternatively, an executable can be created for instant loading (see :doc `save-exec`). The second event (`vescmul-parse`) parses the target design, taking a few seconds. The Verilog file is available in the ACL2 git repository under the books/projects/vescmul/demo directory. The third event (`vescmul-verify`) uses VeSCMul to verify the design. `:concl` specifies the conjecture, with `RESULT` as the output signal name, and `IN1` and `IN2` as input signal names. `logext` sign-extends bit-vectors (represented as integers), and `loghead` zero-extends or, in other words, truncates them. The inputs are 64-bit signed numbers, producing a 128-bit multiplication result. VeSCMul can fully verify this design in 1-2 seconds (as tested on a Macbook M1 pro).

3 Tool Flow

The `vescmul-parse` and `vescmul-verify` utilities are two LISP macros that invoke various programs to parse and then verify target designs.

The `vescmul-parse` macro packs VL/SV/SVTV utilities to parse Verilog designs and create symbolic simulation vectors. These utilities are publicly available and come with the ACL2 installation. They have been developed and used in industry (i.e., Centaur Technology and Intel Corporation) (see `:doc sv`).

The `vescmul-verify` macro gathers the symbolic simulation objects, detects adder components, applies the S-C-Rewriting algorithm, and maybe utilizes SAT solving in the end. The program flow is shown in Fig. 1. These steps are explained as follows.

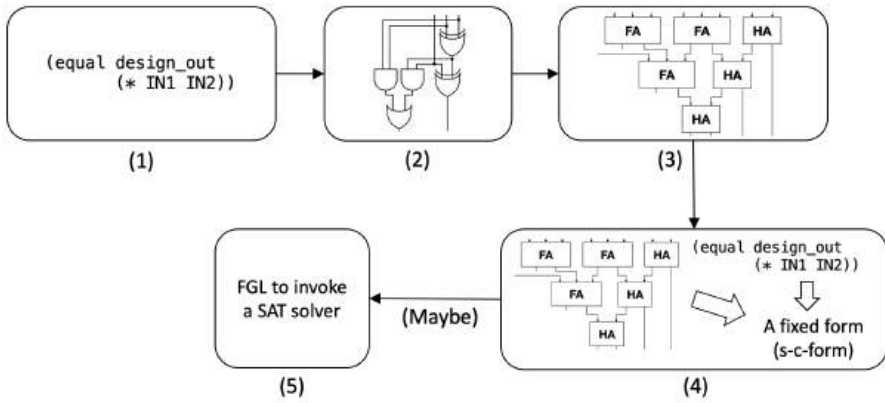


Fig. 1: Flow chart of `vescmul-verify`. (1) User states a conjecture with high-level specification. (2) VeSCMul receives a sea of gates from the design. (3) The tool identifies and rebuilds half/full-adders in this sea of gates. (4) The design and the spec are rewritten with the S-C-Rewriting methodology. (5) If rewriting is not conclusive, rewritten conjecture can be passed to FGL for SAT solving.

(1) Specification is provided by the user, stating a relation between input and output signals. This is typically a combination of multiplication (`*`), addition (`+`), subtraction (`-`), truncation/zero-extension (`loghead`), sign-extension (`logext`), part selection (`part-select`), and possibly user-defined functions.

(2)(3) S-C-Rewriting algorithm needs to differentiate and specially rewrite adder components (e.g., full/half-adders) in a design. In previous work [25,26], S-C-Rewriting algorithm was used only for designs whose design hierarchy information around adders was readily available. VeSCMul has been improved to now support flattened designs. This is achieved by an internal program that goes through a sea of gates to identify and mark the adder components before applying the S-C-Rewriting algorithm. Tests have shown that this program works very well for successful verification of various architectures (see Sec. 5). Should

the program not identify some adders and the verification attempt fails because of that, users may also pass hierarchical verification hints (see `:doc vescmul`).

(4) When VeSCMul applies S-C-Rewriting, the rewriter tries to rewrite both the specification and the design to the same form (i.e., *s-c-form* [25]), and the two sides are compared syntactically. For correct multiplier designs, this is usually enough to prove the conjecture.

(5) If S-C-Rewriting cannot show the conjecture to be correct, it returns its rewritten form. Users have the option to automatically use the FGL utility [19] (see `:doc fgl`) that can bit-blast the rewritten conjecture, perform AIG transformations, and invoke an external SAT solver like CaDiCaL [1]. FGL is also a verified program. This can either generate counterexamples for false conjectures, or help finalize the proofs in some fringe cases. For example, in x86 multiplier designs, extra circuitry is used to calculate flags based on multiplication results, such as the overflow flag that is set when a certain portion of the result are not homogeneously 0s or 1s. VeSCMul by itself may not be able to process the extra flag logic; however, it can rewrite and simplify the multiplication component, send the rewritten expression to an external SAT solver through FGL, and finalize such proofs in a matter of seconds or minutes. Note that if the multiplication component is not rewritten as intended by S-C-Rewriting, it is unlikely for a SAT solver to scale and finish the proofs for operand sizes greater than 16-bits.

4 Notable Features and Compatible Tools

This section highlights some of the useful and noteworthy features of VeSCMul as well as compatible tools.

Customizable specification: Users can state their own specifications to verify various multiplier configurations such as multiply-add, dot product, and multipliers with shifted, truncated and/or saturated outputs.

Automatic adder detection: VeSCMul includes an adder-detection program that identifies and marks adders before employing the S-C-Rewriting algorithm. This makes the overall verification procedure fully automatic for a large variety of multiplier designs (see Sec. 5 for experiments).

End-to-end verified: The author has rigorously verified, using ACL2, that VeSCMul’s all rewriting operations on given conjectures are sound. Users can place high confidence in the results when a design is claimed to be correct. Verifying such a substantial program is a complex process, demanding ACL2 expertise [20,21,22].

Exporting a clean multiplier with design hierarchy: The included adder-detection program can be used as a stand-alone feature. Given a flattened multiplier design, VeSCMul can export a functionally equivalent Verilog module with adder components separated as half/full-adder submodules. This feature may be particularly useful for researchers addressing the multiplier verification problem, where adder detection can be a common challenge [7,11,12,14]. For soundness, VeSCMul includes a mechanism for formal equivalence checking between the original and exported designs.

Integration into other verification flows: Proofs generated by VeSCMul can be integrated into other ACL2-based verification workflows. For instance, when verifying floating-point fused-multiply-add (FMA) operations, which often involves decomposing the design into integer multiplication and post-multiplication parts, VeSCMul can be used for the multiplication part while SAT solving can be employed for the rest. Existing and actively used decomposition tool flows in ACL2 (see `:doc decomposition-proofs`) and VeSCMul are compatible.

Verification of sequential circuits: VeSCMul can handle sequential circuits, including pipelined designs. Additional key arguments can be provided to `vescmul-parse` to verify such designs (see `:doc vescmul-parse`). Modules with control logic reusing the same circuitry for various arithmetic operations (e.g., see `:doc multiplier-verification-demo-2`) are also supported.

Waveform generation: VeSCMul is compatible with another tool (see `:doc svtv-debug$`) for generating waveforms in the VCD format. This capability can be valuable for pinpointing the cause of bugs in case of counterexamples.

5 Experiments

VeSCMul has undergone extensive testing and utilization across various architectures in both public benchmarks and proprietary x86 processor design projects at Centaur Technology and Intel Corporation.

Various benchmarks are gathered for experiments using publicly available generators [3,13,23]¹. Summation trees include Dadda (dt), Wallace (wt), 4-to-2 compressor (4:2), array (ar), redundant binary addition (rbat), balanced delay (bdt), overturned stairs (os) trees. Partial products include signed/unsigned (s/u) simple (sp), Booth radix-2 (b2), radix-4 (r4), radix-8 (r8), radix-16 (r16) encodings. Final stage adders include block carry lookahead (bcla), carry lookahead (cla), carry-select (csel), Ladner-Fischer (lf), carry-skip (csk), conditional sum (cond), Brent-Kung (bk), ripple-carry (rp), Kogge-Stone (ks), Han-Carlson (hc), J. Sklansky conditional (jsk) adders.

Table 1 contains a large number of benchmarks to compare the performance of VeSCMul to other prominent verification tools: AMulet [8] and RevSCA2 [12] that target $n \times n$ -bit multipliers with $2n$ -bit results. The newest version of AMulet (AMulet2) timed out for the majority of the benchmarks; the owner is notified, and AMulet1 is used in the experiments instead. The results for AMulet1 includes the time to check for proof certificates. RevSCA2 is neither a verified program nor does it produce certificates to check its results. These experimental results show that VeSCMul scales much better for large multipliers.

In addition to standard input/output sizes ($n \times n$ -bit multipliers with $2n$ -bit results), Table 2 includes VeSCMul's verification results for variations such as multiply-add (e.g., $64 \times 64 + 64$), multipliers with asymmetric operand sizes (e.g., 10×1024), shifted/truncated outputs (e.g., $64 \times 64[95:32]$ returns the output bit positions from 32 to 95), and dot product (e.g., $8(16 \times 16) + 32$ is an

¹ All tests are available at <https://temelmertcan.github.io/mult-experiments.html>, or the peer-reviewed artifact is available at <https://zenodo.org/records/10048797>

Table 1: Proof-time results with success rates for a large set of nxn-bit multipliers

Op. Size	PP	Benchmarks	RevSCA2 [12]	AMulet1 [7]	VeSCMul
32x32	sp	48	0.5s (77%)	0.4s (100%)	0.5s (100%)
	r2	48	0.8s (62%)	1.4s (100%)	0.7s (100%)
	r4	48	1.4s (87%)	1.3s (91%)	0.6s (100%)
	r8	48	241s (44%)	TO (0%)	0.7s (100%)
	r16	48	TO (0%)	TO (0%)	1.9s (100%)
64x64	sp	54	11s (77%)	1.9s (100%)	1.7s (100%)
	r2	48	17s (62%)	32s (100%)	2.6s (100%)
	r4	240	19s (75%)	4.9s (88%)	2.8s (90%)
	r8	48	1630s (19%)	TO (0%)	2.7s (100%)
	r16	48	TO (0%)	TO (0%)	8s (100%)
128x128	sp	54	83s (65%)	11s (100%)	6.6s (100%)
	r2	48	356s (52%)	928s (100%)	10.1s (100%)
	r4	48	642s (50%)	274s (91%)	8.4s (100%)
	r8	48	TO (0%)	TO (0%)	11s (100%)
	r16	48	TO (0%)	TO (0%)	37s (100%)
256x256	sp	48	2501s (65%)	82s (100%)	27s (100%)
	r4	48	TO (0%)	9529s (91%)	33s (100%)
512x512	r4	6	TO (0%)	TO (0%)	138s (100%)
1024x1024	r4	6	TO (0%)	TO (0%)	776s (100%)

Multiplier sizes range from 32x32 to 1024x1024, grouped wrt. partial product algorithm. Total of 1032 different benchmarks are used and the timing results of successful proof attempts are averaged. The tools could not verify all the benchmarks and the success ratios are given in parentheses. VeSCMul is used only for fully automatic verification (without a SAT solver), but it can verify the missing cases with user-provided hints. Time-out (TO) is set to 3600 seconds (1 hour) for up to 128x128; 16200 seconds (4.5 hours) for the rest. Collected on Intel[®] E-2378G CPU, 32GB memory.

Table 2: Proof-time and memory allocation for various designs

Arch.	Function	Time, Mem		Arch.	Function	Time, Mem
dt-ub4-bcla	64x64	2.1s, 0.3GB		4:2-ub4-cla	64x64	9.7s, 0.7GB
ar-sb4-csel	64x64	2.0s, 0.3GB		rbat-sb4-lf	64x64	2.4s, 0.3GB
bdt-sb4-csk	64x64	2.4s, 0.3GB		os-sb4-cond	64x64	1.8s, 0.3GB
dt-ssp-bk	64x64	1.7s, 0.3GB		ar-usp-rp	64x64	1.0s, 0.2GB
4:2-ub4-ks	64x64	3.7s, 0.5GB		4:2-ub8-lf	64x64	3.4s, 0.5GB
dt-sb16-hc	64x64	8.0s, 1.9GB		wt-ub16-bk	64x64	8.3s, 1.9GB
dt-ssp-bk	128x128	5.9s, 1.0GB		4:2-ub4-hc	128x128	13.3s, 1.8GB
wt-usp-lf	256x256	28s, 4.4GB		dt-sb4-jsk	256x256	27s, 4.4GB
dt-sb4-jsk	512x512	130s, 19GB		dt-sb4-jsk	1024x1024	725s, 83GB
dt-sb4-ks	10x1024	32s, 5.1GB		dt-sb4-ks	1024x10	32s, 5.7GB
dt-sb2-bk	64x64+64	2.5s, 0.4GB		wt-sb4-lf	64x64[63:0]	0.9s, 0.2GB
wt-sb4-lf	64x64[95:32]	1.8s, 0.3GB		wt-sb4-lf	64x64[127:64]	2.2s, 0.4GB
wt-sb8-bk	8(16x16)+32	2.0s, 0.3GB		dt-sb4-ks	4(32x64)+128	5.2s, 1.1GB

8-point dot product with 16-bit operands accumulating onto a 32-bit number). Comparable verification tools do not support these configurations. VeSCMul can fully automatically verify these designs without user hints or SAT solvers.

Moreover, around 7500 different multiplier designs with diverse architectures, operand sizes, operations, truncation, and shifting were randomly generated [23]. Overall, VeSCMul achieved a 98% success rate for fully automatic verification without hints or SAT solvers. The remaining 2% is mostly made up of multipliers with a special 7-to-3 compressor tree, and shifted multipliers, but they could still be verified by VeSCMul with a user-provided design hierarchy hint.

VeSCMul has also proven successful in industrial designs, particularly for Intel x86 instructions with various functional configurations, including multiply-accumulate, dot product, output shifting/truncation, flag calculations based on multiplication results, and saturation. In some cases, the assistance of a SAT solver becomes necessary (for flags and saturation). These designs can be fully verified rapidly and automatically, with results similar to those in the public designs. To the best of the author’s knowledge, VeSCMul is the first tool to achieve comparable verification tasks scalably and automatically.

Additionally, VeSCMul has played a vital role in the verification flow of floating-point multiply and fused-multiply-add operations. Verifying these designs is notably challenging, with no known fully automated verification method. We employ decomposition techniques [5,17], where VeSCMul is used for the multiplication part, significantly reducing manual effort. Complete verification of single and double precision operations can be completed in under an hour.

6 Related Work and Conclusion

AMulet [8], RevSCA2 [12], and DyPoSUB [14] are other state-of-the-art tools for multiplier verification. Like VeSCMul, AMulet prioritizes soundness and can produce proof certificates. In contrast, RevSCA2 and DyPoSUB lack such proofs or mechanisms, and DyPoSUB has been identified as unsound [9]. Additionally, these tools primarily focus on verifying $n \times n$ -bit multipliers with $2n$ -bit results. On the other hand, VeSCMul stands out by offering scalable and automatic verification for a broader range of multiplier-centric arithmetic circuits, and it allows users to specify their conjectures. These target designs can encompass regular multipliers, multiply-add operations, dot products, and operations involving shifting, truncation, accumulation, and saturation.

This paper has showcased VeSCMul for multiplier verification, which has demonstrated favorable results in experiments involving both public and proprietary RTL designs. This tool is open-source and compatible with other hardware verification tools. It has an improved user-interface tailored for ACL2 novices. The tool itself is fully verified, so users can have a high level of confidence in its soundness. Future work includes adding support for more input formats (currently limited to System Verilog) such as AIGER and DIMACS CNF, and further enhancements in automation to handle corner-case designs that currently require user hints for verification.

References

1. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Jarvisalo, M., Suda, M. (eds.) *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
2. Ciesielski, M., Su, T., Yasin, A., Yu, C.: Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019). <https://doi.org/10.1109/tcad.2019.2912944>
3. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Arithmetic module generator (AMG) (2006), <https://www.ecsis.riec.tohoku.ac.jp/views/amg-e>
4. Hunt, W.A., Swords, S., Davis, J., Slobodova, A.: Use of Formal Verification at Centaur Technology. In: Hardin, D. (ed.) *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pp. 65–88. Springer (2010). https://doi.org/10.1007/978-1-4419-1539-9_3
5. Jacobi, C., Weber, K., Paruthi, V., Baumgartner, J.: Automatic formal verification of fused-multiply-add FPUs. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*. p. 1298–1303. DATE '05, IEEE Computer Society, USA (2005). <https://doi.org/10.1109/DATE.2005.75>
6. Kaivola, R., O’Leary, J.: Verification of Arithmetic and Datapath Circuits with Symbolic Simulation, pp. 1–52. Springer Nature Singapore, Singapore (2022). https://doi.org/10.1007/978-981-15-6401-7_37-1
7. Kaufmann, D., Biere, A., Kauers, M.: Verifying Large Multipliers by Combining SAT and Computer Algebra. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. pp. 28–36 (Oct 2019). <https://doi.org/10.23919/FMCAD.2019.8894250>
8. Kaufmann, D., Biere, A.: AMulet 2.0 for verifying multiplier circuits. In: Groote, J.F., Larsen, K.G. (eds.) *International Conference on Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2021. Lecture Notes in Computer Science*, vol. 12652, pp. 357–364. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_19
9. Kaufmann, D., Biere, A.: Fuzzing and delta debugging and-inverter graph verification tools. In: Kovács, L., Meinke, K. (eds.) *Tests and Proofs*. pp. 69–88. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-09827-7_5
10. Kaufmann, M., Moore, J.S.: ACL2 and its applications to digital system verification. In: Hardin, D.S. (ed.) *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pp. 1–21. Springer (2010), https://doi.org/10.1007/978-1-4419-1539-9_1
11. Liew, V., Beame, P., Devriendt, J., Elffers, J., Nordström, J.: Verifying properties of bit-vector multiplication using cutting planes reasoning. In: *2020 Formal Methods in Computer Aided Design (FMCAD)*. pp. 194–204 (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_27
12. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. pp. 185:1–185:6. DAC ’19, ACM, New York, NY, USA (2019). <https://doi.org/10.1145/3316781.3317898>
13. Mahzoon, A., Große, D., Drechsler, R.: SCA multiplier generator GenMul (2019), <https://github.com/amahzoon/genmul>

14. Mahzoon, A., Große, D., Scholl, C., Drechsler, R.: Towards formal verification of optimized and industrial multipliers. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 544–549 (2020). <https://doi.org/10.23919/DATE48585.2020.9116485>
15. Russinoff, D.M.: Formal Verification of Floating-Point Hardware Design: A Mathematical Approach. Springer (2019). <https://doi.org/10.1007/978-3-319-95513-1>
16. Sayed-Ahmed, A., Große, D., Kühne, U., Soeken, M., Drechsler, R.: Formal Verification of Integer Multipliers by Combining Gröbner Basis with Logic Reduction. In: Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1048–1053. Research Publishing Services (2016). https://doi.org/10.3850/9783981537079_0248
17. Slobodová, A.: Challenges for formal verification in industrial setting. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) Formal Methods: Applications and Technology. pp. 1–22. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), https://doi.org/10.1007/978-3-540-70952-7_1
18. Slobodova, A., Davis, J., Swords, S., Hunt, W.A.: A Flexible Formal Verification Framework for Industrial Scale Validation. In: Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 89–97. IEEE/ACM, Cambridge, UK (2011). <https://doi.org/10.1109/memcod.2011.5970515>
19. Swords, S.: New rewriter features in FGL. Electronic Proceedings in Theoretical Computer Science **327**, 32–46 (Sep 2020). <https://doi.org/10.4204/eptcs.327.3>
20. Temel, M.: RP-Rewriter: An optimized rewriter for large terms in ACL2. vol. 327, p. 61–74. Open Publishing Association (Sep 2020). <https://doi.org/10.4204/eptcs.327.5>
21. Temel, M.: Automated, Efficient, and Sound Verification of Integer Multipliers. Ph.D. thesis, The University of Texas at Austin (2021), <https://repositories.lib.utexas.edu/handle/2152/88056>
22. Temel, M.: Verified implementation of an efficient term-rewriting algorithm for multiplier verification on ACL2. International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2022) **359**, 116–133 (may 2022). <https://doi.org/10.4204/eptcs.359.11>
23. Temel, M.: Multgen: a fast multiplier generator (2023), <https://github.com/temelmertcan/multgen>
24. Temel, M.: Formal Verification of Booth Radix-8 and Radix-16 Multipliers. In: Design, Automation & Test in Europe Conference & Exhibition (DATE) (to appear) (2024)
25. Temel, M., Hunt, W.A.: Sound and automated verification of real-world RTL multipliers. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19–22, 2021. pp. 53–62. IEEE (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_13
26. Temel, M., Slobodova, A., Hunt, W.A.: Automated and scalable verification of integer multipliers. In: Computer Aided Verification. pp. 485–507. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_23

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.





A Logical Treatment of Finite Automata

Nishant Rodrigues^(✉), Mircea Octavian Sebe, Xiaohong Chen,
and Grigore Roşu

University of Illinois at Urbana-Champaign, Champaign, USA
{nishant2, osebe2, xc3, grosu}@illinois.edu

Abstract. We present a sound and complete axiomatization of finite words using matching logic. A unique feature of our axiomatization is that it gives a *shallow embedding* of regular expressions into matching logic, and a *logical* representation of finite automata. The semantics of both expressions and automata are precisely captured as matching logic formulae that evaluate to the corresponding language. Regular expressions are matching logic formulae *as is*, while the embedding of automata is a *structural analog*—computational aspects of automata are captured as syntactic features. We demonstrate that our axiomatization is sound and complete by showing that runs of Brzozowski’s procedure for equivalence checking correspond to matching logic proofs. We propose this as a general methodology for producing machine-checkable formal proofs, enabled by capturing structural analogs of computational artifacts in logic. The proofs produced can be efficiently checked by the Metamath Zero verifier. Work presented in this paper contributes to the general scheme of achieving verifiable computing via logical methods, where computations are reduced to logical reasoning, encoded as machine-checkable proof objects, and checked by a trusted proof checker.

1 Motivation

Regular expressions are a powerful lens for studying the description, classification, and implementation of regular languages [14]. A typical presentation of the syntax of extended regular expressions (ERE) over a finite alphabet A is as follows:

$$\alpha := \emptyset \mid \epsilon \mid a \in A \mid \alpha_1 \cdot \alpha_2 \mid \alpha_1 + \alpha_2 \mid \alpha^* \mid \neg\alpha$$

where ϵ is the empty word, $\alpha_1 \cdot \alpha_2$ is concatenation, $\alpha_1 + \alpha_2$ is alternation (aka choice; sum; union), and α^* is the Kleene star. Given a regular expression α , $\mathcal{L}(\alpha)$ is the set of finite words that match α .

A second lens, of finite automata, allows us to view these languages from a *computational* perspective. [14] and [27] show that a language is regular if and only if it is accepted by a finite automaton. Besides providing deeper insight into the study of languages, this dual viewpoint has practical importance—some tasks are easier to tackle when viewed under one lens than another. For example, in the implementation of a parser, it is easier to express the desired language as an expression, whereas an automaton may be used to recognize that language. Model checking [13], and runtime monitoring [3] also exploit these dual perspectives.

Much research has been carried out in the *logical* aspects of regular *expressions*, and the *computational* aspects of finite *automata*. For example, [23] gives an axiomatization of regular expressions in terms of eleven axioms and two inference rules, while automata are used extensively to study complexity theory [13].

In this paper, we instead study *logical* aspects of automata. We present a new axiomatization of finite words using matching logic [8]. This axiomatization gives us a shallow embedding of regular expressions into matching logic where expressions are matching logic formulae *as is*. Uniquely, we can also represent automata as logical formulae. These formulae are a *structural analog* of the automaton—computational aspects such as non-determinism and cycles are captured using syntactic constructs such as logical disjunction and fixpoint operators. We will compare our shallow embedding with prior work using second-order logic, and other formalizations and axiomatizations in Section 2.

Based on our axiomatization, we propose a general technique for generating machine checkable proofs of algorithms that manipulate finite automata. We show that this technique is practical by generating proofs of equivalence between regular expressions from a derivative of Brzozowski’s method [4], producing concrete proofs in matching logic’s proof system realized in Metamath Zero [6]. As touched on in Section 7, an extension to this work may produce proofs for a symbolic execution based compiler [25] allowing us to trust its correctness.

Work presented here contributes to the scheme of verifiable computing [2] via logical methods: computations are reduced to logical reasoning, encoded as machine-checkable proofs, and checked by a small trusted checker, thus reducing our trust-base to the checker while avoiding the expense of full formal verification.

The rest of the paper is organized as follows:

- Section 2 briefly describes prior work in relation to our work.
- Section 3 reviews regular expressions, automata and related concepts.
- Section 4 introduces matching logic and presents a model of finite words.
- Section 5 shows how we may axiomatize this model, and prove equivalent regular expressions and automata.
- Section 6 gives a brief description of our implementation.
- Section 7 lays out some future avenues for research.

Detailed proofs may be found in the companion technical report [21].

2 Related Work

Monadic Second-order-logic (MSO) over Words There is a well-known connection between MSO and regular languages. Büchi, Elgot, and Trakhtenbrot showed that MSO formulae and regular expressions are equally expressive [5, 11, 28]. Moreover, the transformation from expressions to formulae and back is easily computable [26]. Models are sets of labeled positions, representing a word. The set of models that satisfy a formula give us its language—e.g. the MSO formula

\perp defines the empty language—no word satisfies it, while $\exists x. P_a x \wedge \forall y. x = y$ defines the language containing the word a . Here, $P_a x$ indicates the letter a at position x is a . The concatenation of languages may be defined as:

$$\exists X. \forall y, z. ((y \in X \wedge z \notin X) \rightarrow y < z) \wedge [\varphi_\alpha]_{x \in X} \wedge [\varphi_\beta]_{x \notin X}$$

Here, $[\varphi]_{\psi(x)}$ denotes the relativization of the formula φ to the formula ψ , a transformation that forces it to apply to a particular subdomain of the model. The translation of Kleene star is even more complex. This connection has been used, e.g. in the verification of MSO formulae [29].

One concern about this connection between MSO and regular expressions is that the translation of expressions is quite involved, including complex auxiliary clauses and quantification, as well as the relativization transformation. Our goal here is to define a shallow embedding, rather than a translation—regular expressions are directly embedded as matching logic with minimal *representational distance*.

Salomaa’s Axiomatization In [23] Salomaa provides a complete axiomatization of regular expressions that may be used to prove equivalences. This axiomatization is specific to unextended regular expressions and does not support other representations such as negations in EREs, and finite automata.

Deep Embeddings of Automata and Languages There are several existing formalizations of regular expressions and automata using mechanical theorem provers, such as Coq [10] and Isabelle [16]. To the best of our knowledge, all these formalizations use deep embeddings. In [10], the authors formalize regular expressions and Brzozowski derivatives, with the denotations of regular expressions defined using a membership predicate. Besides proving the soundness of Brzozowski’s method, the authors also prove that the process of taking derivatives terminates through a notion of finiteness called *inductively finite sets*. This is something that is not likely provable in a shallow embedding like ours.

Fixpoint Reasoning in Matching Logic We consider our work an extension of the work in [9], where the authors begin tackling the problem of fixpoint reasoning in matching logic. Their goal was to use matching logic as unified framework for fixpoint reasoning across domains. Using a small set of derived matching logic inference rules, they proved various results in LTL, reachability, and separation logic with inductive definitions. We employ many of the techniques first described there, but in addition deal with more complex inductive proofs and recursion schemes, besides producing formal proof certificates.

3 Preliminaries

3.1 Languages, Automata, and Expressions

A language is a set of finite sequences over letters of an alphabet. ERE and finite automata are two ways to represent a class of languages called regular languages.

Definition 1. Let $A = \{a_1, a_2, \dots, a_n\}$ be a finite alphabet. Then ERE over the alphabet A are defined using the following grammar:

$$\alpha := \emptyset \mid \epsilon \mid a \in A \mid \alpha \cdot \alpha \mid \alpha^* \mid \neg\alpha$$

The language that an ERE represents, denoted $\mathcal{L}(\alpha)$ is defined inductively:

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset & \mathcal{L}(\epsilon) &= \{\epsilon\} & \mathcal{L}(a) &= \{a\} \\ \mathcal{L}(\alpha_1 + \alpha_2) &= \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2) \\ \mathcal{L}(\alpha_1 \cdot \alpha_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(\alpha_1) \text{ and } w_2 \in \mathcal{L}(\alpha_2)\} \\ \mathcal{L}(\alpha^*) &= \bigcup_{n=0}^{\infty} \mathcal{L}(\alpha^n) \quad \text{where } \alpha^0 = \epsilon, \text{ and } \alpha^n = \alpha \cdot \alpha^{n-1} \\ \mathcal{L}(\neg\alpha) &= A^* \setminus \mathcal{L}(\alpha) \end{aligned}$$

Since EREs include both complement and choice, other operators like intersection, subsumption and equivalence are definable as notation. We denote these as $\alpha \wedge \beta \equiv \neg(\alpha + \neg\beta)$, $\alpha \rightarrow \beta \equiv \neg\alpha + \beta$, and $\alpha \leftrightarrow \beta \equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$ respectively.

Definition 2. A non-deterministic finite automaton (NFA) is a tuple $\mathcal{Q} = (Q, A, \delta, q_0, F)$, where

- Q is a finite set of states,
- A is a finite set of input symbols called the alphabet,
- $\delta : Q \times A \rightarrow \mathcal{P}(Q)$ is a transition function,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of accepting states.

If $\text{range}(\delta)$ has only singleton sets, \mathcal{Q} is a deterministic finite automaton (DFA).

3.2 Brzowski's Method

In [4], Brzowski introduced an operation over languages called its derivative, denoted $\delta_a(\alpha)$. This operation “consumes” a prefix from each word in the language:

Definition 3. Given a language L and a word s , the derivative of L with respect to s is denoted by $\delta_s(L)$ and is defined as $\{t \mid s \cdot t \in L\}$.

For EREs, it turns out that the derivative can also be defined syntactically, as a recursive function, through the following equalities:

$$\begin{aligned} \delta_a(\epsilon) &= \emptyset & \delta_a(\alpha_1 + \alpha_2) &= \delta_a(\alpha_1) + \delta_a(\alpha_2) \\ \delta_a(\emptyset) &= \emptyset & \delta_a(\alpha_1 \cdot \alpha_2) &= \delta_a(\alpha_1) \cdot \alpha_2 + \alpha_1|_a \cdot \delta_a(\alpha_2) \\ \delta_a(a) &= \epsilon & \delta_a(\alpha^*) &= \delta_a(\alpha) \cdot \alpha^* \\ \delta_a(b) &= \emptyset \quad \text{if } a \neq b. & \delta_a(\neg\alpha) &= \neg\delta_a(\alpha) \\ \delta_\epsilon(\alpha) &= \alpha & \delta_{a \cdot w}(\alpha) &= \delta_w(\delta_a(\alpha)) \end{aligned}$$

Here, $\alpha|_a$ is ϵ if the language of α contains a and \emptyset otherwise. There are two properties of derivatives that are important to us. First, every ERE may be transformed into an equivalent one partitioning its language per the initial letter:

Theorem 1 (Brzowski Theorem 4.4). *Every ERE α can be expressed as:*

$$\alpha = \alpha|_{\epsilon} + \sum_{a \in A} a \cdot \delta_a(\alpha)$$

Second, repeatedly taking the derivative converges:

Theorem 2 (Brzowski Theorem 5.2). *Two EREs are similar iff they are identical modulo associativity, commutativity and idempotency of the $+$ operator. Every ERE has only a finite number of dissimilar derivatives.*

These two properties give rise to an algorithm for converting an ERE into a DFA, illustrated in Figure 1. The automaton is constructed starting from the root node, identifying each node with an ERE. The root node is identified with the original ERE. Every node has transitions for each input letter to the node identified by the derivative. A state is accepting if its language contains the empty word, a property easily checked as a syntactic function of the identifying ERE. This process must terminate by Theorem 2, giving us a DFA. We can check if the ERE is valid by simply checking that all states are accepting.

4 Matching Logic and the Standard Model of Words

In this section, we will review the syntax and semantics of matching logic and present a matching logic model of finite words. We show how it may be used to embed both EREs and finite automata. Matching logic, originally proposed in [22], was revised in [8] to include a fixpoint operator. We present a variant, called polyadic matching logic, omitting sorts since we do not need them¹.

4.1 An Overview of Matching Logic

Matching logic has three parts—a syntax of formulae, also called patterns; a semantics, defining a satisfaction relation \models ; and a Hilbert-style proof system, defining a provability relation \vdash . We will only go over the first two, and then return to matching logic’s proof system in the Section 5.

Syntax Matching logic formulae, or *patterns*, are built from propositional operators, symbol applications, variables, quantifiers, and a fixpoint binder.

Definition 4. *Let EVar , SVar , Σ be disjoint countable sets. Here, EVar contains element variables, SVar contains set variables and the signature $\Sigma = \{\Sigma_n\}$ is an arity-indexed set of symbols. A Σ -pattern over Σ is defined by the grammar:*

$$\varphi := \sigma(\varphi_1, \dots, \varphi_n) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 = \varphi_2 \mid \varphi_1 \subseteq \varphi_2 \mid x \mid \exists x. \varphi \mid X \mid \mu X. \varphi$$

¹ It has since been observed that sorts may be defined axiomatically, and it is unnecessary to build them into the logic. It is called *polyadic* to distinguish it from applicative matching logic with only nullary symbols but includes an explicit application operator.

Note that we have assumed more operators than necessary—equality and subset may be defined in terms of the remaining operators. Please refer to [8] for details. We assume the usual notation for operators such as \top , \vee , \wedge , \forall , ν etc. Here, ν is the greatest fixpoint operator, defined as $\nu X. \varphi \equiv \neg \mu X. \neg \varphi[\neg X/X]$.

Semantics: An Informal Overview Matching logic formulae have a pattern matching semantics. Each pattern φ *matches* a set of elements $|\varphi|$ in the model, called its interpretation. As an example, consider the naturals \mathbb{N} as a model with symbols zero and succ. Here, the pattern \top matches every natural, whereas $\text{succ}(x)$ matches $x + 1$. Conjunctions and disjunctions behave as intersections and unions—the $\varphi \vee \psi$ matches every pattern that either φ or ψ match.

Unlike first-order logic, matching logic makes no distinction between terms and formulae. We may write $\text{succ}(x \vee y)$ to match both $x + 1$ and $y + 1$. While unintuitive at first, this syntactic flexibility allows us to shallowly embed varied and diverse logics in matching logic with ease. Examples include first-order logic, temporal logics, separation logic, and many more [8, 7]. Formulae are embedded as patterns with little to no *representational distance*, quite often verbatim.

Patterns aren't two valued as in first-order logic. We can restore the classic semantics by using the set M to indicate “true” and \emptyset for “false”. The operators $=$ and \subseteq are *predicate patterns*—they are either true or false. For example, $x \subseteq \text{succ}(\top)$ matches every natural if x is non-zero, and no element otherwise. This allows us to build *constrained patterns* of the form $\varphi_{\text{structure}} \wedge \varphi_{\text{constraints}}$. Here, $\varphi_{\text{structure}}$ defines the structure, while $\varphi_{\text{constraints}}$ places logical constraints on matched elements. For example, the pattern $x \wedge (x \subseteq \text{succ}(\top))$ matches x , but only if it is the successor of some element—i.e. non-zero.

Existential quantification works just as in first-order logic when working over predicate patterns. Over more general patterns, it behaves as the union over a set comprehension. For example, the pattern $\exists x. x \wedge (x \subseteq \text{succ}(\top))$ matches *every* non-zero natural. Finally, the fixpoint operator allows us to inductively build sets, as in algebraic datatypes or inductive functions. For example, the pattern $\mu X. \text{zero} \vee \text{succ}(\text{succ}(X))$ defines the set of even numbers.

Semantics: A Formal Treatment We will now formally define the semantics of matching logic. In the interest of brevity we keep things concise. For a more detailed treatment please refer to [8]. Matching logic patterns are interpreted in a model, consisting of a nonempty set M of elements called the universe, and an interpretation $\sigma_M : M^n \rightarrow \mathcal{P}(M)$ for each n -ary symbol $\sigma \in \Sigma$.

Definition 5 (Matching logic semantics). *An M -valuation is a function $\text{EVar} \cup \text{SVar} \rightarrow \mathcal{P}(M)$, such that each $x \in \text{EVar}$ evaluates to a singleton. For a model M and an M -valuation ρ , the interpretation of patterns is defined as:*

$$\begin{aligned}
|x|_M^\rho &= \rho(x), \quad |X|_M^\rho = \rho(X) & |\sigma(\varphi_1, \dots, \varphi_n)|_M^\rho &= \bigcup_{a_i \in |\varphi_i|_M^\rho} \sigma_M(\varphi_1, \dots, \varphi_n) \\
|\neg\varphi|_M^\rho &= M \setminus |\varphi|_M^\rho \\
|\exists x. \varphi|_M^\rho &= \bigcup_{a \in M} |\varphi|_{M, \rho[a/x]}^\rho & |\varphi_1 \vee \varphi_2|_M^\rho &= |\varphi_1|_M^\rho \cup |\varphi_2|_M^\rho \\
& & |\mu X. \varphi|_M^\rho &= \text{lfp} \{A \mapsto |\varphi|_{M, \rho[A/X]}^\rho\} \\
|\varphi_1 \subseteq \varphi_2|_M^\rho &= \begin{cases} M & \text{if } |\varphi_1|_M^\rho \subseteq |\varphi_2|_M^\rho \\ \emptyset & \text{otherwise} \end{cases} & |\varphi_1 = \varphi_2|_M^\rho &= \begin{cases} M & \text{if } |\varphi_1|_M^\rho = |\varphi_2|_M^\rho \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

For the most part, this definition is as expected. For the predicate patterns, the corresponding patterns evaluate to M if they hold, otherwise to the empty set. Besides these, patterns have the obvious evaluation—set and element variables are evaluated according to ρ ; logical operators are evaluated as the corresponding set operation; symbols as defined by the model; existentials as the union for x ranging over M ; and μ as the fixpoint of the interpretation of the pattern.

4.2 A Model of Finite Words

Let us introduce a model \mathcal{M} as the standard model of finite words. Define signature Σ_{Word} containing constants ϵ and a for each $a \in A$, and a binary symbol concat for concatenation. This model allows us to describe languages, including those of regular expressions and finite automata as patterns.

Definition 6. *Let \mathcal{M} be a model for the signature Σ_{Word} with universe the set of finite sequences over alphabet A , and the following interpretations of symbols:*

- $\epsilon_{\mathcal{M}} := \{()\}$,
- for each letter a , $a_{\mathcal{M}} := \{(a)\}$, and
- $\text{concat}_{\mathcal{M}}(s_1, s_2) := \{s_1 \cdot s_2\}$.

Patterns interpreted in model \mathcal{M} define languages. ϵ is interpreted as the singleton set containing the zero-length word, each letter as the singleton set containing the corresponding single-letter sequence, and finally, concat as the function mapping each pair of input words to the singleton containing their concatenation.

We may define the empty language simply as \perp . The concatenation of two patterns gives the concatenation of their languages. Matching logic's disjunction allows us to take the union for any languages, while negation gives us the complement. Finally, we may define the Kleene closure of a language using the fixpoint operator— $\mu X. \epsilon \vee \varphi \cdot X$ gives us the Kleene closure of the language of φ .

A Shallow Embedding of Extended Regular Expressions It is easy to define regular expressions as patterns, once we have the following notation:

$$\emptyset \equiv \perp \quad (\varphi + \psi) \equiv \varphi \vee \psi \quad \varphi^* \equiv \mu X. \epsilon \vee (\varphi \cdot X)$$

Any ERE taken *verbatim* is interpreted in model \mathcal{M} as its language.

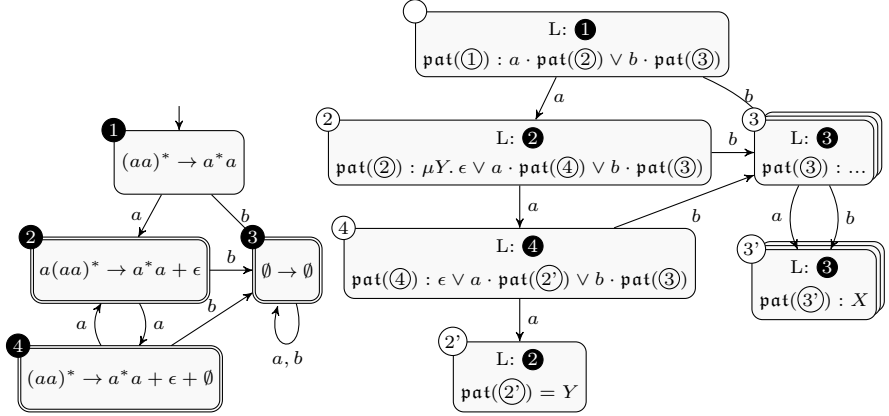


Fig. 1: A DFA \mathcal{Q} for the ERE $(aa)^* \rightarrow a^*a$, and its corresponding unfolding tree. Each node n shows its label $L(n)$, and the pattern $\text{pat}(n)$. Here $\text{pat}(3) \equiv \mu X. \epsilon \vee a \cdot \text{pat}(3) \vee b \cdot \text{pat}(3)$. The pattern for the automaton, $\text{pat}_{\mathcal{Q}}$, is that of the root node $\text{pat}(1)$. Observe that its structure closely mirrors that of \mathcal{Q} . Accepting nodes include ϵ as a disjunct, whereas others do not. Starting a cycle in the graph introduces a fixpoint binder, whereas completing one employs the bound variable corresponding to that cycle. The major structural differences are due to duplicate states to allow backlinks and nodes reachable via multiple paths.

Contrast this to the MSO translation of concatenation, shown in Section 2, and especially of Kleene star.

Theorem 3. *Let α be an ERE. Then $\mathcal{L}(\alpha) = |\alpha|_{\mathbb{W}}$*

4.3 Embedding Automata

While it is obvious how to embed expressions the representation of automata, being computational rather than logical, is less clear. Here, we define a pattern $\text{pat}_{\mathcal{Q}}$ whose interpretation is the language of a finite automaton \mathcal{Q} , either deterministic or non-deterministic. Crucially, this pattern captures not just the language of the automaton (in Section 2 we mentioned that it is possible to do this in MSO as well), but also its *structure*—as shown in Table 1, structural elements of the automata map to syntactic elements of the pattern—non-determinism maps to logical disjunctions; cycles map to fixpoints. This allows us to represent transformations of automata, such as making a transition, union, or complementation, as *logical manipulations* of this pattern in a proof system. This is imperative to capturing the execution of an algorithm employing these in a formal proof. To define $\text{pat}_{\mathcal{Q}}$, we must first define the *unfolding tree* of the automaton \mathcal{Q} .

Definition 7. *For a finite automaton $\mathcal{Q} = (Q, A, \delta, q_0, F)$, its unfolding tree is a labeled tree (N, E, L) where N is the set of nodes, $E \subseteq A \times N \times N$ is a*

Computational aspect of \mathcal{Q}	Syntactic aspect of $\mathbf{pat}_{\mathcal{Q}}$
Node n is accepting	ϵ is a subclause of $\mathbf{pat}(n)$
Non-determinism, union of FAs	Logical union
Graph cycles	Fixpoint binder and its bound variable
Changing the initial node	Unfolding, framing

Table 1: Structural aspects of \mathcal{Q} become syntactic aspects of $\mathbf{pat}_{\mathcal{Q}}$. This is crucial to capturing the traces of algorithms as proofs.

labeled edge relation, and $L : N \rightarrow Q$ is a labeling function. It is the tree defined inductively:

- the root node has label q_0 ,
- if a node n has label q with no ancestors also labeled q , then for each $a \in A$ and $q' \in \delta(q, a)$, there is a node $n' \in N$ with $L(n') = q'$, and $(a, n, n') \in E_a$.

When \mathcal{Q} is a DFA, we use n_a to denote the unique child of node n along edge a . All leaves in this tree are labeled by states that complete a cycle in the automaton. We define a secondary labeling function, $\mathbf{pat} : N \rightarrow \mathbf{Pattern}$ over this tree.

Definition 8. Let (N, E, L) be an unfolding tree for $\mathcal{Q} = (Q, A, \delta, q_0, F)$. Let $X : Q \rightarrow \mathbf{SVar}$ be an injective function. Then, we define \mathbf{pat} recursively as follows:

1. For a leaf node n , $\mathbf{pat}(n) := X(L(n))$.
2. For a non-leaf node,
 - a. if n doesn't have a descendant with the same label, then:

$$\mathbf{pat}(n) = \begin{cases} \epsilon \vee \bigvee_{(a,n,n') \in E} a \cdot \mathbf{pat}(n') & \text{if } L(n) \text{ is accepting.} \\ \bigvee_{(a,n,n') \in E} a \cdot \mathbf{pat}(n') & \text{otherwise.} \end{cases}$$

- b. if n has a descendant with the same label, then:

$$\mathbf{pat}(n) = \begin{cases} \mu X(L(n)). \epsilon \vee \bigvee_{(a,n,n') \in E} a \cdot \mathbf{pat}(n') & \text{if } L(n) \text{ is accepting.} \\ \mu X(L(n)). \bigvee_{(a,n,n') \in E} a \cdot \mathbf{pat}(n') & \text{otherwise.} \end{cases}$$

Finally, define $\mathbf{pat}_{\mathcal{Q}} := \mathbf{pat}(\mathcal{R})$, where \mathcal{R} is the root of this tree.

For nodes of the form (2b), we “name” them by binding the variable $X(L(n))$ using the fixpoint operator. When we return to that state we use the bound variable to complete a cycle. The use of fixpoints allows us to clearly embody the inductive structure as a pattern. Figure 1 shows an example of unfolding tree. The following theorem shows that this representation of automata is as expected.

Theorem 4. Let \mathcal{Q} be a finite automaton. Then $\mathcal{L}(\mathcal{Q}) = |\mathbf{pat}_{\mathcal{Q}}|_{\mathbf{w}}$

4.4 Embedding Brzowski's Derivative

Besides regular languages, other important constructs may be defined using this model. Let us look at derivatives, needed to capture Brzowski's method as a proof. The Brzowski derivative of a language L w.r.t. a word w , is the set of words obtainable from a word in L by removing the prefix w . Defining this is quite simple in matching logic—for any word w and pattern ψ , we may define its Brzowski derivative as the pattern $\delta_w(\psi) \equiv \exists x. x \wedge (w \cdot x \subseteq \psi)$.

This definition is quite interesting because it closely parallels the embedding of separation logic's magic wand in matching logic: $\varphi \multimap \psi \equiv \exists x. x \wedge (\varphi * x \subseteq \psi)$. At first glance, this seems like a somewhat weak connection, but on closer inspection, magic wand and derivatives are semantically quite similar—we may think of magic wand as taking the derivative of one heap with respect to the other.

It is these connections between seemingly disparate areas of program verification that matching logic seeks to bring to the foreground. In fact, both derivatives and magic wand generalize to a matching logic operator called *contextual implication*: $C \multimap \psi \equiv \exists \square. \square \wedge (C[\square] \subseteq \psi)$ for any pattern ψ and application context C [9]. Using this notation, derivatives and magic wand become $\delta_w(\varphi) \equiv w \cdot \square \multimap \varphi$ and $\varphi \multimap \psi \equiv \varphi * \square \multimap \psi$ respectively. This operator has proven key to many techniques for fixpoint reasoning in matching logic, especially the derived rules (WRAP) and (UNWRAP) that enable applying Park induction within contexts [9]:

$$\vdash C[\varphi] \rightarrow \psi \quad \frac{(\text{UNWRAP})}{(\text{WRAP})} \quad \vdash \varphi \rightarrow (C \multimap \psi)$$

5 Proof Generation

In the previous section, we showed how we may capture languages as matching logic patterns. Specifically, automata are captured as patterns that are structural analogs. In this section, we will demonstrate how we capture runs of algorithms that manipulate automata as proofs. In particular, we capture runs of Brzowski's method using matching logic's Hilbert style proof system.

This technique is only possible because of the structural similarity between an automata \mathcal{Q} , and its pattern $\text{pat}_{\mathcal{Q}}$. It gives us the ability to represent computational transformations on automata as *logical* transformations of these patterns using matching logic's proof system. This section focuses on the theory and proofs involved. The subsequent section, Section 6, will present our concrete implementation producing matching logic proofs that can be checked using Metamath Zero. Let us first introduce matching logic's proof system, and a theory Γ_{Word} within which we do our reasoning.

5.1 Matching Logic's Proof System

The third component to matching logic is its proof system, shown in Figure 2. It defines the provability relation, written $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system using the theory Γ as additional axioms.

(PROPOS. 1)	$\varphi \rightarrow (\psi \rightarrow \varphi)$	(PROPAG $_{\perp}$)	$C[\perp] \rightarrow \perp$
(PROPOS. 2)	$(\varphi \rightarrow (\psi \rightarrow \theta))$ $\rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \theta))$	(PROPAG $_{\vee}$)	$C[\varphi \vee \psi] \rightarrow C[\varphi] \vee C[\psi]$
(PROPOS. 3)	$((\varphi \rightarrow \perp) \rightarrow \perp) \rightarrow \varphi$	(PROPAG $_{\exists}$)	$C[\exists x. \varphi] \rightarrow \exists x. C[\varphi]$ where $x \notin FV(C)$
(MP)	$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$	(FRAMING)	$\frac{\varphi \rightarrow \psi}{C[\varphi] \rightarrow C[\psi]}$
<hr/>			
(\exists -QUANT.)	$\varphi[y/x] \rightarrow \exists x. \varphi$	(\exists -GEN.)	$\frac{\varphi \rightarrow \psi}{(\exists x. \varphi) \rightarrow \psi}$ where $x \notin FV(\psi)$
<hr/>			
(PRE-FP)	$\varphi[(\mu X. \varphi)/X] \rightarrow \mu X. \varphi$	(KT)	$\frac{\varphi[\psi/X] \rightarrow \psi}{(\mu X. \varphi) \rightarrow \psi}$
<hr/>			
(EXISTENCE)	$\exists x. x$	(SUBST)	$\frac{\varphi}{\varphi[\psi/X]}$
(SINGLETON)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$		

Fig. 2: Matching logic proof system. Here C, C_1, C_2 are application contexts, a pattern in which a distinguished element variable \square occurs exactly once, and only under applications. We use the notation $C[\varphi] \equiv C[\varphi/\square]$.

These proof rules fall into four categories. First, the FOL rules provide complete FOL and propositional reasoning. The (PROPAGATION) rules allow applications to commute through constructs with a “union” semantics, such as disjunction and existentials. The proof rule (KNASTER-TARSKI) is an embodiment of the Knaster-Tarski fixpoint theorem [24], and together with (PREFIXEDPOINT) correspond to the Park induction rules of modal logic [18, 15]. Finally, (EXISTENCE), (SINGLETON), and (SUBST) are technical rules, needed to work with variables.

5.2 A Theory of Finite Words

We may use a theory Γ , a set of patterns called *axioms*, to restrict the models we consider to those in which every axiom is “true”. We say a pattern φ *holds* in a model M , or that φ is *valid* in M , written $M \models \varphi$ if its interpretation is M under all evaluations. For a theory Γ , we write $M \models \Gamma$ if every axiom in Γ is valid in M . For a pattern ψ , we write $\Gamma \models \psi$ if for every model where $M \models \Gamma$ we have $M \models \psi$. These axioms also extend the provability relation \vdash defined by the proof system, allowing us to proof additional theorems. The soundness of matching logic guarantees that each proved theorem holds in every model of the theory.

Figure 3 defines a theory, Γ_{Word} , of finite words. The first set of the axioms in Γ_{Word} , (FUNC $_{\sigma}$), gives each symbol a functional interpretation: for an n -ary symbol σ , the axiom $\forall x_1, \dots, x_n. \exists y. \sigma(x_1, \dots, x_n) = y$, forces the interpretation σ_M to return a single output for any input. This is because element variables are always interpreted as singleton sets. Next, the (NO-CONF) axioms ensure that

Signature: ϵ , \cdot , $_$, and a for each $a \in A$.

Axioms:

For each $a \in A$,

For each distinct $a, b \in A$,

$\exists w. a = w$	(FUNC _a)	$a \neq b$	(NO-CONF _a)
$\exists w. \epsilon = w$	(FUNC _ε)	$\epsilon \not\subseteq a \vee b$	(NO-CONF _ε)
$\forall u, v. \exists w. u \cdot v = w$	(FUNC _•)	$\forall u, v. \epsilon = u \cdot v \rightarrow$	
$\forall u, v, w. (u \cdot v) \cdot w = u \cdot (v \cdot w)$	(ASSOC)	$u = \epsilon \wedge v = \epsilon$	(NO-CONF _• -1)
$\forall x. (\epsilon \cdot x) = x$	(ID _L)	$\forall x, y : \text{Letter}. \forall u, v.$	
$\forall x. (x \cdot \epsilon) = x$	(ID _R)	$x \cdot u = y \cdot v \rightarrow x = y \wedge u = v$	(NO-CONF _• -2)
		$\mu X. \epsilon \vee \bigvee_{a \in A} a \cdot X$	(DOMAIN)

Fig. 3: Γ_{Word} : A theory of finite words in matching logic. This theory is complete for proving equivalence between representations of both automata and extended regular expressions. Here, $\forall x : \text{Letter}. \varphi$ is notation for $\forall x. x \in (\bigvee_{a \in A} a) \rightarrow \varphi$.

interpretations of symbols are injective modulo AU—they have distinct interpretations unless their arguments are equal modulo associativity of concatenation with unit ϵ . Here, $\forall x : \text{Letter}. \varphi$ is notation for $\forall x. x \in (\bigvee_{a \in A} a) \rightarrow \varphi$, i.e. we quantify over letters. The axioms (ASSOC), and (ID_L), and (ID_R) enforce the corresponding properties and allow their use in proofs. The final axiom (DOMAIN) defines our domain to be inductively constructed from ϵ , concatenation and letters. It is easy to see the standard model satisfies these axioms, giving us the theorem, proved in the appendix [21]:

Theorem 5. $\models \Gamma_{\text{Word}}$

The rest of this section is dedicated to showing that Γ_{Word} is complete with respect to both equivalence of automata and EREs—if two automata or expressions have the same language their representations are provably equivalent.

5.3 Proving Equivalence between EREs

We are now ready to demonstrate our proof generation method. We will use it to capture equivalence of expressions using matching logic’s Hilbert-style proof system. Brzozowski’s method consists of two parts—converting an ERE into a DFA \mathcal{Q} , and checking that \mathcal{Q} is total. Mirroring this, the proof for equivalence between EREs $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$ has two parts. First, we prove that $\Gamma_{\text{Word}} \vdash \text{pat}_{\mathcal{Q}} \rightarrow (\alpha \leftrightarrow \beta)$ —the language of $\alpha \leftrightarrow \beta$ subsumes that of \mathcal{Q} . Second, that $\Gamma_{\text{Word}} \vdash \text{pat}_{\mathcal{Q}}$ —the language of \mathcal{Q} is total. We put these together using (MODUS-PONENS), giving us $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$ —the EREs are provably equivalent.

Proving $\Gamma_{\text{Word}} \vdash \text{pat}_{\mathcal{Q}} \rightarrow (\alpha \leftrightarrow \beta)$ To prove this, we prove a more general, inductive, lemma:

Lemma 1. *Let n be a node in the unfolding tree of the DFA \mathcal{Q} of the regular expression $\alpha \leftrightarrow \beta$, where α and β have the same language. Then,*

$$\Gamma \vdash \mathbf{pat}(n)[\Lambda_n] \rightarrow \delta_{\mathbf{path}(n)}(\alpha \leftrightarrow \beta)$$

where,

$$\Lambda_n = \begin{cases} \lambda, \text{ the empty substitution} & \text{if } n \text{ is the root node} \\ \Lambda_p[\delta_{\mathbf{path}(p)}(\alpha \leftrightarrow \beta)/X(p)] & \text{if } n \text{ has parent } p, \text{ and } \mathbf{pat}(p) \text{ binds } X(p) \\ \Lambda_p & \text{otherwise.} \end{cases}$$

The substitution Λ_n provides the inductive hypothesis—as we use the (KNASTER-TARSKI) rule on each μ -binder in $\mathbf{pat}_{\mathcal{Q}}$, it replaces the bound variable with the right-hand side of the goal. The left-hand side then becomes a disjunction of the form $\epsilon \vee a \cdot \mathbf{pat}(n_a)[\Lambda_{n_a}] \vee b \cdot \mathbf{pat}(n_b)[\Lambda_{n_b}]$. We decompose the right-hand side into a similar structure using an important property of derivatives, proved in Γ_{Word} :

Lemma 2. *For any pattern φ , $\Gamma_{\text{Word}} \vdash \varphi = ((\epsilon \wedge \varphi) \vee \bigvee_{a \in A} a \cdot \delta_a(\varphi))$*

The derivatives are reduced to expressions using proved syntactic simplifications:

Lemma 3. *For EREs α, β and distinct letters a and b , the following hold:*

- $\Gamma_{\text{Word}} \vdash \delta_a(\emptyset) = \emptyset$; $\Gamma_{\text{Word}} \vdash \delta_a(\epsilon) = \emptyset$;
- $\Gamma_{\text{Word}} \vdash \delta_a(b) = \emptyset$; $\Gamma_{\text{Word}} \vdash \delta_a(a) = \epsilon$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\alpha_1 + \alpha_2) = \delta_a(\alpha_1) + \delta_a(\alpha_2)$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\alpha_1 \cdot \alpha_2) = \delta_a(\alpha_1) \cdot \alpha_2 + (\alpha_1 \wedge \epsilon) \cdot \delta_a(\alpha_2)$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\neg \alpha) = \neg \delta_a(\alpha)$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\alpha^*) = \delta_a(\alpha) \cdot \alpha^*$.

Proving $\Gamma_{\text{Word}} \vdash \mathbf{pat}_{\mathcal{Q}}$ The next part of the proof is a bit more technical, requiring us to exploit the equivalence $\Gamma_{\text{Word}} \vdash (\mu X. \epsilon \vee X \cdot \varphi) \leftrightarrow (\mu X. \epsilon \vee \varphi \cdot X)$, and induct using the (DOMAIN) axiom. This reduces our goal to $\Gamma_{\text{Word}} \vdash \mathbf{pat}_{\mathcal{Q}} \cdot (\bigvee_{a \in A} a) \rightarrow \mathbf{pat}_{\mathcal{Q}}$, a consequence of the following inductive lemma:

Lemma 4. *Let n be a node in the unfolding tree of a total DFA \mathcal{Q} . Then,*

$$\Gamma_{\text{Word}} \vdash \mathbf{pat}(n)[\Theta_n] \cdot (\bigvee_{a \in A} a) \rightarrow \mathbf{pat}(n)[U_n]$$

where,

$$\Theta_n = \begin{cases} \lambda, \text{ the empty substitution} & \text{if } n \text{ is the root node} \\ \Theta_p[\Psi_p/X_{L(p)}] & \text{if } n \text{ has parent } p, \text{ and } \mathbf{pat}(p) \text{ binds } X(p) \\ \Theta_p & \text{otherwise} \end{cases}$$

$$\Psi_p = \square \cdot (\bigvee_{a \in A} a) \multimap \mathbf{pat}(p)[U_p]$$

$$U_n = \begin{cases} \lambda, \text{ the empty substitution} & \text{if } n \text{ is the root node} \\ U_p[\mathbf{pat}(p)[U_p]/X_{L(p)}] & \text{if } n \text{ has parent } p, \text{ and } \mathbf{pat}(p) \text{ binds } X(p) \\ U_p & \text{otherwise} \end{cases}$$

```

def checkValid( $\varphi$ : Regex, prev: set[Regex] =  $\emptyset$ )  $\rightarrow$  bool:
    if  $\varphi \in \text{prev}$ : return True
    if  $\neg \text{hasEWP}(\varphi)$ : return False
    return checkValid(canonicalize( $\delta_a(\varphi)$ , prev  $\cup$  { $\varphi$ }))
        and checkValid(canonicalize( $\delta_b(\varphi)$ , prev  $\cup$  { $\varphi$ }))

```

Fig. 4: The algorithm instrumented to generate proofs. The `canonicalize` function reduces the pattern $\delta_a(\varphi)$ to an ERE, simplifying it to a form where choice is left-associative, and the idempotency and unit identities have been applied.

Again, Θ_n gives us the inductive hypothesis, this time in the form of a contextual implication. To apply it, we leverage a general property about contextual implications: $\vdash C[C \multimap \varphi] \rightarrow \varphi$, allowing us combine framing with Park induction. This gives us our main theorem, showing that our axiomatization is complete with respect to extended regular expressions:

Theorem 6. *For any EREs α and β with the same language, $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$.*

5.4 From Expressions to Automata

Our uniform treatment of automata and expressions as patterns allows us to apply Brzozowski’s method not just to EREs but also to more general patterns. For example, it can be used to determinize NFAs, or take the complement, union, or intersection of DFAs. The general principle is the same as above, except instead of $\alpha \leftrightarrow \beta$, we use a pattern corresponding to the operation we wish to perform. For example, to prove that the DFA \mathcal{Q} has the same language as the intersection of those of \mathcal{A} and \mathcal{B} , we prove $\Gamma_{\text{Word}} \vdash \text{pat}_{\mathcal{Q}} \leftrightarrow (\text{pat}_{\mathcal{A}} \wedge \text{pat}_{\mathcal{B}})$. All we need is the ability to take the derivative of arbitrary fixpoint patterns enabled by the equivalence $\Gamma_{\text{Word}} \vdash \delta_a(\mu X. \varphi) \leftrightarrow \delta_a(\varphi[\mu X. \varphi/X])$.

6 Implementation and Evaluation

In this section, we describe the implementation of our method. The algorithm implemented is shown in Figure 4. It recursively checks that the expression and its derivatives have the empty-word property, keeping track of when it has already visited an expression. Here, $\delta_a(\varphi)$ represents a *pattern* using the derivative notation, and not the fully simplified regular expression. This notation is simplified away in the (also instrumented) `canonicalize` function that also normalizes the choice operator to be left-associative and commutes subterms into lexicographic order, allowing the application of the idempotency and unit axioms. This results in a canonical representation of expressions modulo similarity.

The instrumentation of successful runs of this method produces a *proof-hint*, an example of which is shown in Figure 5. A proof-hint is an informal artifact

```

(der (a + b)*,
  a : (simpl  $\delta_a((a + b)^*)$ ,
    (simpl  $(\delta_a((a + b)) \cdot (a + b)^*$ 
    (simpl  $(\delta_a(a) + \delta_a(b))(a + b)^*$ 
    (simpl  $(\epsilon + \delta_a(b))(a + b)^*$ 
    (simpl  $(\epsilon + \perp)(a + b)^*$ 
    (simpl  $\epsilon \cdot (a + b)^*$ 
    (backlink (a + b)*)))))),
  b : (simpl  $\delta_b((a + b)^*, \dots)$ 
    der-*,  $\square, \alpha \mapsto (a + b); l \mapsto a$ ,
    der- $\vee$ ,  $\square \cdot (a + b)^*, \dots$ ,
    der-same-letter,  $(\square + \delta_a(b)) \cdot (a + b)^*, \dots$ ,
    der-diff-letter  $(\epsilon + \square) \cdot (a + b)^*, \dots$ ,
    choice-identity-right,  $\square \cdot (a + b)^*, \dots$ ,
    concat-identity-left,  $\square, \dots$ ,

```

Fig. 5: An snippet of a proof-hint for expression $(a + b)^*$ produced by the instrumentation. Most substitutions are omitted for brevity. The lemma `id der-*` corresponds to the metamath theorem for $\Gamma_{\text{Word}} \vdash \delta_l(\alpha^*) = \delta_l(\alpha) \cdot \alpha^*$

containing all the information necessary to produce a formal proof. It is a term defined by the following grammar.

```

Node := (backlink Pattern)
      | (der Pattern, a : Node, b : Node)
      | (simpl Pattern, LemmaID, Context, Subst, Node)

```

These terms are more detailed structures than unfolding trees—if we ignore the simplification nodes, we get an unfolding tree. Each `backlink` and `der` node is labeled by a regular expression, and correspond to the leaf and interior nodes of an unfolding tree. In addition, `der` nodes have child nodes labeled by the patterns $\delta_a(\varphi)$ and $\delta_b(\varphi)$. Note that these are patterns and *not* regular expressions—they use the matching logic notation for derivative, and are distinct from the fully simplified EREs. Each `simpl` node keeps track of equational simplifications needed to reduce the derivative notation, and employs associativity, commutativity, and idempotency of choice to reduce the expression into a canonical form, allowing the construction of unfolding tree to terminate. The `simpl` nodes contain the name of the simplification applied, the context in which it was applied, as well as the substitutions with which it was applied. The `LemmaID` corresponds to a hand-proved lemma in the Metamath Zero formalization.

To produce the proof of validity, proof-hints are used in three contexts. First, to produce the pattern `patQ`; next, to produce an instance of Lemma 1; and finally, to produce an instance of Lemma 4. For each lemma, we inductively build up the proof from two manually proven Metamath Zero theorems, one for the `backlink` node case, and another for the `der` node case. In the case of Lemma 1, the `simpl` nodes are ignored. In the case of Lemma 4 we use them to reduce the patterns to their canonical form. This is done by lifting a manually

Benchmark	Nodes	.mmb size	Gen. time	Check time
Manual Lemmas		307		3
$(a + b)^*$	3	2	64	3
$a^{**} \rightarrow a^*$	5	4	82	3
$(aa)^* \rightarrow a^*a + \epsilon$	9	15	179	3
$\neg(\top \cdot a \cdot \top) + \neg(b^*)$	5	5	90	3
$\text{match}_l(2) / \text{match}_l(8)$	19 / 43	13 / 266	273 / 27483	3 / 4
$\text{match}_r(2) / \text{match}_r(8)$	19 / 43	13 / 228	337 / 21085	3 / 4
$\text{eq}_l(2) / \text{eq}_l(8)$	13 / 37	15 / 446	374 / 91661	3 / 5
$\text{eq}_r(2) / \text{eq}_r(8)$	13 / 37	15 / 330	368 / 31489	3 / 5

Table 2: Statistics for certificate generation. Sizes are in KiB, times in milliseconds. We show the unfolding tree nodes, proof size, generation and checking time.

proven theorem corresponding to the LemmaId into the context, and applying the substitution, all supplied by the `simpl` node.

Trust Base Our trust base consists of the Metamath Zero formalization of matching logic proof system, including its syntax and meta-operations for its sound application such as substitution, freshness (272 lines); the theory of words instantiated with $A = \{a, b\}$, (13 lines); and the Metamath Zero proof checker, `mm0-c`. Each of these are defined in `.mm0` files in our repository [20, 19]. From these, we prove by hand 354 supporting general theorems and 163 specific to Γ_{Word} , such as Lemmas 1 and 4, and those about derivatives and their simplification.

Evaluation We have evaluated our work against handcrafted tests, as well as standard benchmarks for deciding equivalence presented in [17]. Some statistics are shown in Table 2. Each $\text{match}_{\{l,r\}}(n)$ test, by [12], is an ERE asserting that a^n matches $(a + \epsilon) \cdot a^n$, that is, $a^n \rightarrow (a + \epsilon) \cdot a^n$. Here α^n indicates n -fold concatenation of α , with the l version using concatenation from the left, and the r version on the right. That is, α^3 may be either $((\alpha \cdot \alpha) \cdot \alpha)$ or $(\alpha \cdot (\alpha \cdot \alpha))$. Each $\text{eq}_{\{l,r\}}(n)$ test, by [1], checks if a^* and $(a^0 + \dots + a^n) \cdot (a^n)^*$ are equivalent. We also include property testing using the Hypothesis testing framework. We randomly generate an ERE α , and check that $\alpha \rightarrow \alpha$. Our procedure does not optimize for this, so it allows testing correctness for a variety of expressions, augmenting the few handcrafted ones, and the structurally monotonous benchmarks.

Performance In this work, our goal was to prove that this process is feasible—we have not focused on performance. In fact, we find the performance numbers here are quite poor. There are a number of reasons for this.

First, we made some poor implementation choices with reference to instrumentation. The prototype uses Maude and its meta-level to produce the instrumentation. While Maude’s search command *collects* all the information needed for the proof hint, it does not make it accessible. This forced us to repeatedly enter and exit

the meta-level to collect this information, bringing the running time of, e.g., $\text{match}_l(8)$ to 27 seconds, compared to 3ms when implemented idiomatically.

Another reason is that we targeted simplicity, rather than even the most basic optimizations. For example, when multiple identical nodes occur in an unfolding tree, we do not reuse the subproofs for identical nodes in the derivative tree, and instead re-prove the result each time. This causes a significant blow up in proof size. We believe that a relatively small engineering effort would greatly improve performance both in terms of proof size and generation time.

Another issue is that handling machine generated proofs is not one of Metamath Zero's design goals. It is intended as a human-readable language, for human-written proofs. We would rather output a succinct binary representation of proofs. Although Metamath Zero does allow generation of proofs directly in the mmb format, this seems closer to an embedded systems format than a formal language.

7 Future Work and Conclusion

Study of Languages Definable in Γ_{Word} While this paper has focused on regular languages in Γ_{Word} , we can define more languages. For example, the context-free language $\{a^n \cdot b^n \mid n \in \mathbb{N}\}$ may be defined as $a^n \cdot b^n \equiv \mu X. \epsilon \vee a \cdot X \cdot b$. Extending this, we may define $a^n \cdot b^n \cdot c^i$, and $a^i \cdot b^n \cdot c^n$ for $n, i \in \mathbb{N}$ as the patterns $a^n \cdot b^n \cdot c^*$ and $a^* \cdot b^n \cdot c^n$ respectively. Finally, since patterns are closed under intersection we may define the *context-sensitive* language $a^n \cdot b^n \cdot c^n \equiv (a^n \cdot b^n \cdot c^*) \wedge (a^* \cdot b^n \cdot c^n)$. Extensive research has been done regarding languages definable in fragments of MSO. A corresponding effort for matching logic would be interesting. Likely, quantifiers and fixpoint operators will allow defining most computable languages.

Application to Control Flow Graphs (CFGs) Through the \mathbb{K} Framework, the transition systems of programming languages are defined in matching logic. The CFGs of programs in these languages may be viewed as automata. Our technique would allow formal proofs of correctness of algorithms over the CFGs of programs, such as the semantics-based compiler in [25].

Acknowledgements We warmly thank Mario Carneiro for his invaluable feedback on the usage of Metamath Zero. We are indebted to the anonymous reviewers for their kind input and suggestions.

References

- [1] Valentin Antimirov. “Partial derivatives of regular expressions and finite automata constructions”. In: *STACS 95: 12th Annual Symposium on Theoretical Aspects of Computer Science Munich, Germany, March 2–4, 1995 Proceedings*. Springer. 2005, pp. 455–466.
- [2] Konstantine Arkoudas and Selmer Bringsjord. “Computers, justification, and mathematical knowledge”. In: *Minds and Machines* 17 (2007), pp. 185–202.
- [3] Ezio Bartocci et al. “Introduction to runtime verification”. In: *Lectures on Runtime Verification: Introductory and Advanced Topics* (2018), pp. 1–33.
- [4] Janusz A Brzozowski. “Derivatives of regular expressions”. In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.
- [5] J. Richard Buchi. “Weak Second-Order Arithmetic and Finite Automata”. In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92. DOI: <https://doi.org/10.1002/malq.19600060105>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19600060105>.
- [6] Mario Carneiro. “Metamath Zero: Designing a theorem prover prover”. In: *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings 13*. Springer. 2020, pp. 71–88.
- [7] Xiaohong Chen and Grigore Roşu. “A general approach to define binders using matching logic”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–32.
- [8] Xiaohong Chen and Grigore Roşu. *Matching μ -logic*. Tech. rep. <http://hdl.handle.net/2142/102281>. University of Illinois at Urbana-Champaign, Jan. 2019.
- [9] Xiaohong Chen et al. “Towards a unified proof framework for automated fixpoint reasoning using matching logic”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.
- [10] Thierry Coquand and Vincent Siles. “A decision procedure for regular expression equivalence in type theory”. In: *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7–9, 2011. Proceedings 1*. Springer. 2011, pp. 119–134.
- [11] Calvin C Elgot. “Decision problems of finite automata design and related arithmetics”. In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51.
- [12] Sebastian Fischer, Frank Huch, and Thomas Wilke. “A play on regular expressions: functional pearl”. In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 2010, pp. 357–368.
- [13] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. “Introduction to automata theory, languages, and computation”. In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

- [14] SC Kleene. “Representation of events in nerve nets and finite automata”. In: *Automata Studies: Annals of Mathematics Studies. Number 34* 34 (1956), p. 3.
- [15] Dexter Kozen. “Results on the propositional μ -calculus”. In: *Theoretical computer science* 27.3 (1983), pp. 333–354.
- [16] Alexander Krauss and Tobias Nipkow. “Proof pearl: Regular expression equivalence and relation algebra”. In: *Journal of Automated Reasoning* 49 (2012), pp. 95–106.
- [17] Tobias Nipkow and Dmitriy Traytel. “Unified decision procedures for regular expression equivalence”. In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014. Proceedings* 5. Springer. 2014, pp. 450–466.
- [18] David Park. “Fixpoint induction and proofs of program properties”. In: *Machine intelligence* 5 (1969).
- [19] Nishant Rodrigues and Mircea Sebe. *A Logical Treatment of Finite Automata (Artifact)*. Dec. 2023. DOI: 10.5281/zenodo.10431211. URL: <https://doi.org/10.5281/zenodo.10431211>.
- [20] Nishant Rodrigues and Mircea Sebe. *Matching Logic in MM0*. Oct. 2023. URL: <https://github.com/formal-systems-laboratory/matching-logic-in-mm0> (visited on 04/14/2023).
- [21] Nishant Rodrigues et al. *Technical Report: A Logical Treatment of Finite Automata*. Tech. rep. <https://hdl.handle.net/2142/121770>. 2024.
- [22] Grigore Roşu. “Matching Logic”. In: *Logical Methods in Computer Science* Volume 13, Issue 4 (Dec. 2017). DOI: 10.23638/LMCS-13(4:28)2017. URL: <https://lmcs.episciences.org/4153>.
- [23] Arto Salomaa. “Two complete axiom systems for the algebra of regular events”. In: *Journal of the ACM (JACM)* 13.1 (1966), pp. 158–169.
- [24] Alfred Tarski et al. “A lattice-theoretical fixpoint theorem and its applications.” In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309.
- [25] The K Team. *KSummarizer*. 2022. URL: <https://research.runtimeverification.com/#the-k-summarizer> (visited on 10/16/2023).
- [26] Wolfgang Thomas. “Languages, automata, and logic”. In: *Handbook of Formal Languages: Volume 3 Beyond Words*. Springer, 1997, pp. 389–455.
- [27] Ken Thompson. “Programming Techniques: Regular Expression Search Algorithm”. In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: <https://doi.org/10.1145/363347.363387>.
- [28] Boris Avraamovich Trakhtenbrot. “Finite automata and the logic of one-place predicates”. In: *Sibirskii Matematicheskii Zhurnal* 3.1 (1962), pp. 103–131.
- [29] Dmitriy Traytel and Tobias Nipkow. “Verified decision procedures for MSO on words based on derivatives of regular expressions”. In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 3–12.

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.





A State-of-the-Art Karp-Miller Algorithm Certified in Coq

Thibault Hilaire^(✉), David Ilcinkas, and Jérôme Leroux

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France
`{thibault.hilaire,david.ilcinkas,jerome.leroux}@labri.fr`

Abstract. Petri nets constitute a well-studied model to verify and study concurrent systems, among others, and computing the coverability set is one of the most fundamental problems about Petri nets. Using the proof assistant Coq, we certified the correctness and termination of the MINCOV algorithm by Finkel, Haddad, and Khmelnitsky (FOSSACS 2020). This algorithm is the most recent algorithm in the literature that computes the minimal basis of the coverability set, a problem known to be prone to subtle bugs. Apart from the intrinsic interest of a computer-checked proof, our certification provides new insights on the MINCOV algorithm. In particular, we introduce as an intermediate algorithm a small-step variant of MINCOV of independent interest.

Keywords: Petri net · Karp-Miller tree algorithm · Minimal coverability set · Coq · Certified decision procedure

1 Introduction

Petri nets constitute a well-studied model to verify and study concurrent systems, with several applications in other domains, like in chemical [1] and biological process [2,26] (see [31] for additional applications). Formally, a Petri net is given by a finite set of *places* and a finite set of *transitions*. Each place is marked with a natural number that can be incremented or decremented by the transitions. A function that maps places to the marked numbers is called a *marking*. The *reachability set* of a Petri net from an initial marking is the set of markings that can be obtained by executing a sequence of transitions from the initial marking.

The central problem about Petri nets is the reachability problem that consists in deciding whether a final marking is in the reachability set. Many important computational problems in logic and complexity reduce or are even equivalent to this problem [15,31]. The reachability problem is known to be Ackermann-complete [5,23,6,20]. On positive instances, it can be decided with efficient directed exploration strategies [3], but general complete algorithms deciding the problem are complex [24], and require a lot of implementation efforts [7].

This high complexity is not always a barrier in practice since many problems related to Petri nets can be decided by introducing an over-approximation of the

reachability set, called the *coverability set* [18]. This set is defined by introducing the cover relation over the markings, defined by $x \leq y$ if x is less than or equal to y component-wise, i.e., on each place. The coverability set is then defined as the downward-closure of the reachability set. It provides a way to decide a variant of the reachability problem, called the coverability problem. This latter problem can be solved by computing what is called a *basis* of the coverability set. Its definition uses the notion of ω -markings, an extension of markings that allows to mark places with a special symbol denoted by ω , and interpreted as an infinite number. The well-quasi-order theory [11] shows that any downward-closed set of markings can be symbolically represented by a finite set of ω -markings, called a *basis*. Moreover, this theory also proves that there exists a unique minimal one for the inclusion relation.

The computation of bases of coverability sets is exactly the purpose of the Karp-Miller algorithm introduced in [19]. This algorithm inductively computes trees where nodes are labeled by ω -markings. When the algorithm stops, those labels form a basis of the coverability set. Karp-Miller algorithms (including all variants) are not optimal in worst-case complexity for deciding the coverability problem. In fact, those algorithms have an Ackermannian computational complexity [8,25] while the coverability problem is known to be Expspace-complete [28]. There exist other algorithms, based on backward computations from the final marking, that are optimal in worst-case [4,21]. However, Karp-Miller algorithms outperform backward computation algorithms in practice (see [3] for benchmarks). Moreover, the computation of the coverability set bases provides ways to decide other properties than the coverability problem, like the termination and boundedness problems, as well as some liveness properties. It follows that this algorithm is central for analyzing Petri nets.

Bases computed by the Karp-Miller algorithm are not minimal (for the inclusion relation) since they may contain distinct ω -markings x, y with $x \leq y$. Naturally, the unique minimal basis of the coverability set can be computed by first invoking the Karp-Miller algorithm, and then applying a simple reduction algorithm. However, such a computation is not optimal in practice since it requires computing several ω -markings that will be discarded only at the end of the computation. A first attempt to avoid this problem was introduced by Alain Finkel in [9]. This algorithm is an optimization of the original Karp-Miller algorithm that seems very natural. However, a subtle problem when the computation is performed on a very particular instance was discovered only 14 years later in [10]. Several authors tried to find patches for that bug by proposing various solutions [13,29,27,30]. Finally, in [12], an efficient algorithm removing on-the-fly useless basis elements was proved to be correct with a pen-and-paper proof. This algorithm, called MINCOV, is a state-of-the-art algorithm for computing the minimal basis of the coverability set. It can be seen as a variant of the Karp-Miller algorithm based on the new notions of abstractions and accelerations. Since algorithms à la Karp-Miller are prone to subtle bugs, formal proofs certified by proof assistants are called for.

Our Contributions.

- We developed a complete formal proof in COQ of the correctness and termination of the MINCOV algorithm, via an intermediate algorithm called ABSTRACTMINCOV. We follow the COQ formalization of Petri nets and markings introduced in [33], built on top of the MATHEMATICAL COMPONENTS library [14] (MATHCOMP). This formalization contains several formal proofs and basic concepts related to Petri nets and markings that we extended to handle recent notions. Our proofs are based on this code to take benefits from those developments, but also to easily measure the gap between COQ formal proofs of two algorithms that compute coverability set bases: the original Karp-Miller algorithm and a state-of-the-art one.
- We provide two new characterizations of the central notion of abstractions used by the MINCOV algorithm. A simple mathematical one, and an algebraic one that shows that three operators on abstractions (*weakening*, *contraction*, and *acceleration*) provide a complete set of rules for generating any abstraction starting from the Petri net transitions. The proof of this result is based on the Jančar well-quasi-order on executions [17,22].
- We introduce as an intermediate algorithm a small-step variant of MINCOV, called ABSTRACTMINCOV. We implemented in COQ proofs of the correctness and termination of ABSTRACTMINCOV. Since the original MINCOV algorithm can be simulated by our algorithm, the proof that the original MINCOV algorithm is correct and terminates is obtained at the cost of a simple COQ proof. Compared to a direct proof, our approach provides more succinct proofs in COQ, because proving that some properties are invariant is usually easier for a small step than for a big step. Additionally, our algorithm provides room for optimization by decorrelating some transformations performed by the original algorithm (this is discussed in the conclusion).

Outline. Our COQ formalization of Petri nets, markings, and ω -markings are given in Section 2, while the ones on abstractions and accelerations are given in Section 3. The COQ modelization of MINCOV is provided in Section 4, and our small-step algorithm ABSTRACTMINCOV is presented in Section 5. The code is available on Software Heritage [16].

2 Petri Nets

A *Petri net* is a tuple $\mathcal{P} = \langle P, T, \text{Pre}, \text{Post} \rangle$ where P, T are two finite sets of elements called respectively *places* and *transitions*, and Pre, Post are two mappings from T to \mathbb{N}^P . An element $x \in \mathbb{N}^P$ is called a *marking*. We denote by $x(p)$ the value of x at the place p . Markings $\text{Pre}(t)$ and $\text{Post}(t)$, where t is a transition in T are called respectively the *precondition* and the *postcondition* of t .

We follow the COQ formalization of Petri nets and markings introduced in [33]. That formalization was introduced to prove the correctness and termination of the original Karp-Miller algorithm. This formalization is built on top of the

MATHEMATICAL COMPONENTS library [14] (MATHCOMP). This library provides finite types (see the COQ keyword `finType` below) that provides a useful type for Petri net places and transitions, but also functions with finite domain (see `ffun`). Markings are conveniently represented by these functions. More precisely, in our COQ proofs, Petri nets and markings are defined as follows.

```
Record petri_net :=
  PetriNet
{ place transition : finType;
  _ _ : transition -> {ffun place -> nat}; (* pre, post *)
}.

Definition marking (pn : petri_net) := {ffun place pn -> nat}
(* Re-type the 3rd and 4th fields of PN to use the name "marking". *)

Definition pre (pn : petri_net) : transition pn -> marking pn :=
  let: PetriNet _ _ p _ := pn in p.

Definition post (pn : petri_net) : transition pn -> marking pn :=
  let: PetriNet _ _ _ p := pn in p.
```

Now, let us provide some elements of Petri net semantics. Given a Petri net \mathcal{P} , a transition $t \in T$ is said to be *fireable* from a marking x if $\text{Pre}(t) \leq x$; where \leq is the component-wise extension of the usual order \leq on \mathbb{N} , i.e. $x \leq m$ iff $x(p) \leq m(p)$ for every place $p \in P$. In that case we write $x \xrightarrow{t} y$ where $y = x - \text{Pre}(t) + \text{Post}(t)$ is called the marking obtained after *firing* t from x . We extend the notion of fireability to a sequence $\sigma = t_1 \dots t_k$ of transitions $t_1, \dots, t_k \in T$ by $x \xrightarrow{\sigma} y$ if there exists a sequence x_0, \dots, x_k of markings such that $x_0 = x$, $x_k = y$ and $x_{j-1} \xrightarrow{t_j} x_j$ for every $1 \leq j \leq k$. In that case, we say that σ is *fireable* from x and y is naturally called the marking obtained after *firing* σ from x . When such a sequence σ exists, we say that y is *reachable* from x (for the Petri net \mathcal{P}).

The *Petri net reachability problem* consists in deciding, given a Petri net \mathcal{P} and two markings x, y , whether y is reachable from x . The reachability problem is Ackermann-complete [5,23,6,20] and algorithms deciding the problem are complex [24]. However, this high lower bound is not always a barrier in practice since many problems related to Petri nets can be decided by computing an over-approximation of the reachability property, called the *coverability*, obtained by introducing the downward-closed sets.

More formally, the *downward closure* of a set M of markings is defined as the set $\{x \in \mathbb{N}^P \mid \exists y \in M, x \leq y\}$. We say that M is *downward-closed* if it is equal to its downward closure. Downward-closed sets can be finitely represented by introducing the notion of ω -markings, a notion also known as the *ideal representation* of downward-closed sets (see [11] for extra results). We first introduce the set \mathbb{N}_ω defined as $\mathbb{N} \cup \{\omega\}$, where ω is a special symbol not in \mathbb{N} that is interpreted as an infinite number. This interpretation is defined by extending the total order \leq over \mathbb{N} into a total order on \mathbb{N}_ω by $n \leq \omega$ for every $n \in \mathbb{N}_\omega$. An

ω -marking is an element of $x \in \mathbb{N}_\omega^P$. In [33] and in our COQ proofs, ω -markings are defined with the type `markingc` as follows.

Definition `natc` := `optiontop nat`.

(Here None (also denoted Top) denotes Omega and Some n denotes n *)*

Definition `markingc` := `{ffun place -> natc}`.

We associate with an ω -marking x the downward-closed set $\downarrow x$ of markings defined as $\{y \in \mathbb{N}^P \mid y \leq x\}$. We also denote by $\downarrow B$, where B is a finite set of ω -markings, the downward-closed set $\bigcup_{x \in B} \downarrow x$. Let us recall from the well-quasi-order theory [11] that any downward-closed set M of markings admits a finite set B of ω -markings, called a *basis* of M , such that $M = \downarrow B$. Bases provide finite descriptions of downward-closed sets. Naturally a downward-closed set can have several bases. However, among all the bases of a downward-closed set, the unique minimal one (for the inclusion relation) can be computed from any basis as follows. We say that a finite set B of ω -markings forms an *antichain* if for every $x, y \in B$ such that $x \leq y$, we have $x = y$. Notice that if B is a basis of a downward-closed set M that is not an antichain, then there exist $x, y \in B$ such that $x < y$. Since in that case $B \setminus \{x\}$ is also a basis of M , it follows that by recursively removing from B the ω -markings that are strictly smaller than another one in B , we derive from any basis another one that is an antichain. One can prove that this antichain is the unique minimal basis of M (for the inclusion relation).

Given a Petri net \mathcal{P} , we say that a marking $z \in \mathbb{N}^P$ is *coverable* from a marking x_0 if there exists a marking $y \geq z$ reachable from x_0 . The set of coverable markings is called the *coverability set*.

Since coverability sets are downward-closed, they can be described by bases. The computation of such those bases is exactly the purpose of Karp-Miller algorithms. While ω components were introduced in the original Karp-Miller algorithm [19] with some algorithmic techniques, this notion was abstracted away in [12] as kind of *meta-transitions*, called *accelerations* and *abstractions*. Those notions are recalled in the next section. They are used to compute the minimal basis of the coverability set, called the *clover* in [12]. In our COQ proofs, we encode the clover as a list of ω -markings (a list is denoted by `seq`). The definition uses the `coverable` predicate defined in [33].

Definition `clover` (`m0` : `marking`) (`l` : `seq markingc`) :=

`antichain l /\`

`forall m : marking,`

`coverable m0 m <-> exists mc : markingc, (mc \in l) && (m \in mc).`

(perm_eq is the list equivalence modulo permutation *)*

Theorem `clover_unique` `m0` (`l1 l2` : `seq markingc`):

`clover m0 l1 -> clover m0 l2 -> perm_eq l1 l2.`

3 Abstractions and Accelerations

Abstractions provide a simple way to explain why some markings can be covered from other ones. In this section we first recall the definition and semantics of ω -transitions. Then we introduce the abstractions following the definition introduced in [12], based on ω -transitions. We show that this rather technical definition is in fact equivalent to a new simpler one. Whereas the proof of equivalence between the two definitions is simple, we think that our definition provides interesting intuitions on abstractions. Finally, in the last part of this section we show that three operators on abstractions (*weakening*, *contraction*, and *acceleration*) provides a complete set of rules for generating any abstraction starting from the Petri net transitions. The proof is based on the Jančar well-quasi-order on executions [17,22].

Since our COQ proofs for this part are obtained by series of case analyses (not complicated but lengthy in COQ), we do not provide additional information concerning that part of our implementation. All proofs can be found in the file `New_transitions.v`.

3.1 ω -Transitions

An ω -transition t is a pair $t = (x, y)$ where $x, y \in \mathbb{N}_\omega^P$ are ω -markings such that $x(p) = \omega \Rightarrow y(p) = \omega$ for every place $p \in P$. The ω -markings x and y are respectively denoted by $\text{Pre}(t)$ and $\text{Post}(t)$ and they are called respectively the *precondition* and the *postcondition* of t . This notation provides a natural way to identify transitions of a Petri net as particular ω -transitions. We implemented ω -transitions in COQ with the dependent datatype `omega_transition` as follows.

Definition `transitionc` := (markingc * markingc)%type

(* $t.\text{pre} = \text{Pre}(t)$ and $t.\text{post} = \text{Post}(t)$ *)

Definition `inv_omega_transition` (t: `transitionc`) :=

[forall p , (t.pre p == None) ==> (t.post p == None)].

Definition `omega_transition` := { t | inv_omega_transition t }.

We introduce the operator $\ominus : \mathbb{N}_\omega^P \times \mathbb{N}_\omega^P \rightarrow \mathbb{N}_\omega^P$ defined component-wise by $x \ominus y = 0$ if $x \leq y$, ω if $x = \omega$ and $y \in \mathbb{N}$, and $x - y$ otherwise. As expected, an ω -transition t is said to be *fireable* from an ω -marking x if $\text{Pre}(t) \leq x$. In that case, we write $x \xrightarrow{t} y$ where $y = (x \ominus \text{Pre}(t)) + \text{Post}(t)$ is called the ω -marking obtained after *firing* t from x .

In order to provide a way to manipulate a sequence of ω -transitions as just one single ω -transition, the notion of *Hurdle* [15], known by the Petri net community for sequences of transitions, was extended to sequences of ω -transitions [12]. More formally, we introduce an internal binary operator \otimes on ω -transitions, called the *contraction*, as follows:

$$s \otimes t = ((\text{Pre}(t) \ominus \text{Post}(s)) + \text{Pre}(s) , (\text{Post}(s) \ominus \text{Pre}(t)) + \text{Post}(t))$$

We implemented in COQ the contraction operator and we formally proved the following lemma.

Lemma 1. *For every ω -markings $x, z \in \mathbb{N}_\omega^P$, the ω -transition $s \otimes t$ satisfies:*

$$x \xrightarrow{s \otimes t} z \iff \exists y \in \mathbb{N}_\omega^P, x \xrightarrow{s} y \xrightarrow{t} z$$

In the sequel, given a sequence of ω -transitions $\sigma = t_1 \dots t_k$, we call the ω -transition $t = t_1 \otimes \dots \otimes t_k$ the *contraction* of σ and, when there is no ambiguity, we identify σ with its contraction. It follows that $\text{Pre}(\sigma)$ and $\text{Post}(\sigma)$ are well defined.

3.2 Abstractions

Following [12], an *abstraction* is an ω -transition a such that for all $n \geq 0$, there exists $\sigma_n \in T^*$ such that for all $p \in P$ with $\text{Pre}(a)(p) \in \mathbb{N}$:

- $\text{Pre}(\sigma_n)(p) \leq \text{Pre}(a)(p)$
- If $\text{Post}(a)(p) \in \mathbb{N}$ then $\text{Post}(a)(p) + \text{Pre}(\sigma_n)(p) \leq \text{Post}(\sigma_n)(p) + \text{Pre}(a)(p)$
- If $\text{Post}(a)(p) = \omega$ then $\text{Pre}(\sigma_n)(p) + n \leq \text{Post}(\sigma_n)(p)$

Our COQ implementation of abstractions is a direct translation of the previous definition. We provide the code just below. In that code, note that `seq_to_one` is a function that maps sequences of transitions to their contractions. Also, we provide a simplification of the actual code in which we use the same symbols for comparisons and operations independently of whether `nat`, `natc`, or a mix of the two, are used. Similarly, we assume in the sequel implicit coercions from `omega_transition`, `abstraction`, or `acceleration` to `transitionc`.

```
Definition inv_abstraction_aux (t : transitionc) (y : marking*marking)
  (p : place) (n : nat) :=
  mem_nc (t.pre p) (y.pre p)
  /\ (t.post p != None -> t.post p + y.pre p <= t.pre p + y.post p)
  /\ (t.post p == None -> y.pre p + n <= y.post p).
```

```
Definition inv_abstraction (t : transitionc) :=
  forall (n : nat), exists (o_n : seq transition), forall (p : place),
  t.pre p != None -> (inv_abstraction_aux t (seq_to_one o_n) p n).
```

```
Definition abstraction := { a : omega_transition | inv_abstraction a }.
```

The previous definition of abstraction is in fact equivalent to the following simpler one, where $\text{Cover}(x, \mathcal{P})$ for some ω -marking x denotes the set of markings z such that $x \xrightarrow{\sigma} z$ for some word σ of transitions and some ω -marking $y \geq z$.

Lemma 2. *A given ω -transition a is an abstraction if, and only if, it satisfies $\downarrow \text{Post}(a) \subseteq \text{Cover}(\text{Pre}(a), \mathcal{P})$.*

Note that this new characterization provides a way to constructively check whether an ω -transition is an abstraction. This would allow us to declare abstractions as an `eqType` in a future work.

We also recall the following lemma proved in [12]. This result is central for the correctness of the algorithm MINCOV. We implemented its proof in COQ in the file `New_transitions.v`.

Lemma 3 (Lemma 1 in [12]). *Let x_0 be a marking of a Petri net \mathcal{P} . For every ω -markings x, y such that $x \xrightarrow{a} y$ for some abstraction a , we have:*

$$\downarrow x \subseteq \text{Cover}(x_0, \mathcal{P}) \Rightarrow \downarrow y \subseteq \text{Cover}(x_0, \mathcal{P})$$

3.3 Abstraction Builder

In this last part, we show that any abstraction can be built from Petri net transitions by applying three operators: weakening, contraction, and acceleration.

Let us first start with the simplest operator, called the *weakening*. We introduce a partial order \sqsubseteq on the ω -transitions defined by $s \sqsubseteq t$ if $\text{Pre}(t) \leq \text{Pre}(s)$ and $\text{Post}(s) + \text{Pre}(t) \leq \text{Post}(t) + \text{Pre}(s)$. The second inequality intuitively means that the effect of t is larger than or equal to the effect of s (component-wise). Based on Lemma 2, we deduce that if t is an abstraction and s an ω -transition such that $s \sqsubseteq t$, then s is also an abstraction. Based on this observation, we introduce a weakening operator that just replaces an abstraction t by any other abstraction $s \sqsubseteq t$.

The second simplest operator is the contraction. Based on Lemmas 1 and 2, we can deduce that if s, t are two abstractions, then $s \otimes t$ is also an abstraction.

The last operator, called the *acceleration*, associates with an ω -transition t the ω -transition t^ω that intuitively corresponds to the infinite firing of t . More formally, t^ω is defined as follows for every place $p \in P$:

$$\begin{aligned} \text{Pre}(t^\omega)(p) &= \begin{cases} \omega & \text{if } \text{Pre}(t)(p) > \text{Post}(t)(p) \\ \text{Pre}(t)(p) & \text{otherwise} \end{cases} \\ \text{Post}(t^\omega)(p) &= \begin{cases} \omega & \text{if } \text{Pre}(t)(p) \neq \text{Post}(t)(p) \\ \text{Post}(t)(p) & \text{otherwise} \end{cases} \end{aligned}$$

In [12], it is proved that if a is an abstraction then a^ω is also an abstraction.

Notice that $t^\omega = t$ if, and only if, $\text{Post}(t)(p) \in \{\text{Pre}(t)(p), \omega\}$ for every $p \in P$. If a is an abstraction and $a^\omega = a$, we say that a is an *acceleration*. Since accelerations play a central role in the MINCOV algorithm, we implemented them in COQ as follows.

```
Definition inv_accel (t : transitionc) :=
  [forall p, (t.post p == None) || (t.post p == t.pre p)].
```

```
Definition acceleration := { a : abstraction | inv_accel a }.
```

The following Lemma 4 is one of the main result of this section. It shows that any abstraction can be derived from the Petri net transitions by applying the previously mentioned operators.

Lemma 4. *An ω -transition a is an abstraction if, and only if, there exist $w_0, t_1, w_1, \dots, t_k, w_k$ where $w_0, \dots, w_k \in T^*$ and $t_1, \dots, t_k \in T$ such that:*

$$a \sqsubseteq w_0^\omega t_1 w_1^\omega \dots t_k w_k^\omega$$

4 The Original MINCOV Algorithm

In this section, we present our COQ implementation of the MINCOV algorithm. We tried to be as close as possible to the algorithm introduced in [12], to provide convincing evidence that it is correct and terminating. We however omitted the `trunc` function used in the MINCOV pseudocode presented in [12] but not in their PYTHON implementation. In practice this function differs from the identity function only when numbers computed by the algorithm are larger than the number of atoms in the universe.

4.1 Explicit Coverability Trees

As already mentioned, this algorithm computes the minimal basis of the coverability set of a Petri net \mathcal{P} from an initial ω -marking x_0 . Similarly to the original Karp-Miller algorithm, it computes inductively a tree \mathcal{T} such that nodes are labeled by ω -markings, and edges by transitions. In the case of MINCOV, the constructed tree, called an *explicit coverability tree*, contains additional labels that are explained a bit later. We implement explicit coverability trees in COQ as the following inductive definition KMTE:

```
Inductive KMTE := | Empty_E
                  | Br_E of markingc &
                      (seq acceleration) &
                      bool &
                      {ffun transition -> KMTE}.
```

A node obtained with the constructor `Empty_E` is called *empty*, whereas a node obtained with the constructor `Br_E` is called *valid*. The first line of the constructor `Br_E` of a valid node N provides the ω -marking denoted by $\lambda(N)$ that labels the node N . The fourth line provides a function that inductively maps each transition t to a subtree. The root node of that subtree is denoted by $N.t$ and called the *child* of N following t . Given a node, we call the unique word $\sigma \in T^*$ that labels the edges of the tree from the root to that node the *address* of that node. A word $\sigma \in T^*$ is called a *valid address* if it is the address of a valid node. This node is denoted by N_σ in that case. A node is called a *leaf* if it is valid and if $N.t$ is an empty node for every transition t .

Compared to trees computed by the Karp-Miller algorithm, explicit coverability trees computed by the MINCOV algorithm have two additional pieces of information on each valid node, provided by the second and third lines of the constructor `Br_E`. First of all, since trees may be partially destroyed when a subtree corresponding to redundant computations is detected, the computation is no longer a DFS exploration. In order to keep track of nodes that are waiting for further exploration, called *front nodes*, each valid node is marked with a boolean flag that is assigned to true when it is a front one. The *set of front nodes* of an explicit coverability tree \mathcal{T} is denoted by $\text{Front}(\mathcal{T})$. Last but not least, explicit coverability trees contain additional information to recover the way the node labels were generated. To do so, the second line of the constructor `Br_E` of a valid node N provides a sequence $a_1 \dots a_k$ of accelerations denoted by $\mu(N)$.

In our implementation, we prove that the following properties (called *invariant properties* in the sequel) are maintained throughout any execution of the algorithm.

- Front nodes are always leaves (predicate `Front_leaves`).
- Non-front node labels form an antichain (predicate `Not_Front_Antichain`).
- The root node is valid, and $x_0 \xrightarrow{\mu(N_\varepsilon)} \lambda(N_\varepsilon)$ (predicate `consistentE_head`).
- If a valid node N is not the root, i.e. $N = N'.t$ for some node N' and some transition t , then $\lambda(N') \xrightarrow{t\mu(N)} \lambda(N)$ (predicate `consistentE_tree`).

4.2 Step Relation

The MINCOV algorithm is a **while** loop algorithm that updates a pair (\mathcal{T}, A) , where \mathcal{T} is an explicit coverability tree, and A is a (finite) sequence of accelerations. Accelerations that occur in \mathcal{T} (in the μ labeling) are taken from A . Moreover, the sequence A can only grow with new discovered accelerations. Initially, the MINCOV algorithm begins with the pair (\mathcal{T}, A) where A is the empty sequence ε and \mathcal{T} is the explicit coverability tree reduced to a single valid front node N_ε labeled by $\lambda(N_\varepsilon) = x_0$ and $\mu(N_\varepsilon) = \varepsilon$. The algorithm picks nondeterministically a front node at each iteration of the **while** loop to transform the tree. It terminates when the set of front nodes is empty and, at that point, returns the current \mathcal{T} (the set A is discarded at the end). Our COQ implementation of this algorithm is defined by introducing a binary relation `Rel` on those pairs (\mathcal{T}, A) . Such a one-step encoding provides all the possible nondeterministic behaviors of the algorithm. It follows that our proofs of correctness and termination are valid whatever the implemented particular exploration heuristic.

Formally, the relation `Rel` is defined as follows, with three constructors `Rel_clean`, `Rel_accel`, and `Rel_explo` that are defined later in this section:

Variant `Rel` :

```
(KMTE * seq acceleration) -> (KMTE * seq acceleration) -> Prop :=
| Rel_clean [...] (* cleaning operation *)
| Rel_accel [...] (* accelerating operation *)
| Rel_explo [...] (* exploring operation *).
```

As will be discussed later, the termination of the MINCOV algorithm is proved by certifying that the relation Rel is well-founded. For that reason, $\text{Rel} \ (T', A') \ (T, A)$ corresponds to a step of the MINCOV algorithm from (T, A) to (T', A') , and not the other way around.

One central notion of the algorithm is the definition of saturated ω -markings. An ω -marking x is *saturated* for a sequence A of accelerations if, for every acceleration $a \in A$ such that $x \xrightarrow{a} y$ for some ω -marking y , we have $x = y$. When an ω -marking is not saturated for a sequence A , it can be saturated with respect to A as follows. Note that in general, given two ω -markings x, y such that $x \xrightarrow{a} y$ for some acceleration a , then $y(p) \in \{x(p), \omega\}$ for every place p . It means that y is obtained from x by setting to ω some places of x . In particular, if $x \neq y$, then the number of places with natural numbers is strictly decreasing from x to y . It follows that an algorithm that tries to apply in a round-robin fashion all the accelerations in A eventually terminates on a fixed point in at most $|P|$ rounds. We implement this algorithm in COQ with a function `saturate_KMTree` $A \ T \ \text{ad}$ that takes as input a sequence A of accelerations, an explicit coverability tree \mathcal{T} , and a valid address $\sigma \in T^*$ (denoted by `ad`), and returns the explicit coverability tree obtained from \mathcal{T} by saturating $\lambda(N_\sigma)$ with respect to A , and by appending to $\mu(N_\sigma)$ the sequence of accelerations used by the round-robin saturation algorithm.

The MINCOV algorithm is implemented in such a way the labels of the non-front valid nodes form an antichain. To enforce that property, the *cleaning operation* takes as input two explicit coverability trees \mathcal{T} and \mathcal{T}' , a sequence A of accelerations, and an address σ (denoted by `ad` below), and checks if σ is the address of a front node, if \mathcal{T}' is the tree obtained from \mathcal{T} by saturating N_σ with respect to A (see above), and if there exists a non-front node N' such that $\lambda(N_\sigma) \leq \lambda(N')$ in \mathcal{T}' (predicate `ad_covered_not_front` $T' \ \text{ad}$ below). In that case, the cleaning operation puts in the relation Rel the pair (\mathcal{T}, A) with (\mathcal{T}'', A) , where \mathcal{T}'' is obtained from \mathcal{T}' by removing the node at address σ (implemented by `removeE_add` $T' \ \text{ad}$).

```

Rel_clean (T:KMTE) A ad T': Is_Front T ad
-> T' = saturate_KMTree A T ad
-> ad_covered_not_front T' ad
-> Rel (removeE_add T' ad, A) (T, A)

```

When the previous cleaning operation cannot be applied on a front node with address σ (\sim denotes the negation, and `ad` and `ad'` in the code refer to σ and σ'), the algorithm checks if this front node, once saturated, is labeled by an ω -marking larger than the label of an ancestor with address σ' (through the predicate `Possible_acceleration`, which also checks that σ' is the prefix of σ). If so, an *accelerating operation* is performed. It consists first in computing the acceleration corresponding to the path between the two nodes. More precisely, `computingE_acceleration` $T' \ \text{ad}' \ \text{ad}$ computes the acceleration $a = (t_1\sigma_1 \dots t_k\sigma_k)^\omega$, where $\sigma = \sigma't_1 \dots t_k$ for a sequence $t_1 \dots t_k$ of transitions, and $\sigma_1, \dots, \sigma_k$ are the sequences of accelerations that occur in \mathcal{T}' from σ

to σ' , i.e. $\sigma_j = \mu(N_{\sigma' t_1 \dots t_j})$. In that case, the accelerating operation puts in the relation **Rel** the pair (\mathcal{T}, A) with (\mathcal{T}'', A') , where A' is the sequence obtained by adding a to A , and \mathcal{T}'' is obtained from \mathcal{T}' by removing the subtree of \mathcal{T}' from $N_{\sigma'}$ and by setting that node as a front node (**to_FrontE** T' **ad** below).

```

Rel_accel (T:KMTE) A ad T' ad' a: Is_Front T ad
-> T' = saturate_KMTree A T ad
-> ~~ ad_covered_not_front T' ad
-> Possible_acceleration T' ad' ad
-> a = computingE_acceleration T' ad' ad
-> Rel (to_FrontE T' ad', a :: A) (T,A)

```

When the previous cleaning and accelerating operations cannot be applied on a front node (tested through **No_Possible_acc** for the accelerating operation), the algorithm performs an exploration from that front node by trying to fire all the transitions from the label of that node. This label x is computed after saturation via the function **m_from_add**, from the tree and the address σ (denoted by **ad** below) of the node. The *exploring operation* (see **Rel_explo** below) puts in the relation **Rel** the pair (\mathcal{T}, A) with (\mathcal{T}''', A) , where \mathcal{T}''' is the tree obtained from \mathcal{T}' by removing valid nodes labeled by an ω -marking smaller than x (implemented by **removeE_strict_covered** T' x), and \mathcal{T}''' is obtained from \mathcal{T}'' by removing the node at address σ from the front list, and by creating, for each transition t such that there exists an ω -marking y such that $x \xrightarrow{t} y$, a front node $N_{\sigma t}$ labeled by $\lambda(N_{\sigma t}) = y$ and $\mu(N_{\sigma t}) = \varepsilon$ (this last operation is implemented by **Front_extensionE**).

```

Rel_explo (T:KMTE) A ad T' mc: Is_Front T ad
-> T' = saturate_KMTree A T ad
-> ~~ ad_covered_not_front T' ad
-> No_Possible_acc T' ad
-> Some mc = m_from_add T' ad
-> Rel (Front_extensionE (removeE_strict_covered T' mc) ad, A) (T,A)

```

5 The ABSTRACTMINCOV Algorithm

The COQ proofs of correctness and termination of the MINCOV algorithm are obtained by introducing a variant of that algorithm, called ABSTRACTMINCOV. This new algorithm takes a small-step approach obtained by decomposing the three main operations (cleaning, accelerating, and exploring) of the original MINCOV into sequences of five small-step operations presented in this section.

We implemented in COQ a formalization of ABSTRACTMINCOV and proved the correctness and termination of that algorithm. Since the original MINCOV algorithm can be simulated by our algorithm, we obtain at the cost of a simple COQ proof of simulation that the original MINCOV algorithm is correct and

terminates. Compared to a direct proof, our approach provides more succinct proofs in CoQ, because proving that some properties are invariant is usually easier for a small step than for a big step.

Compared to the original MINCOV algorithm, which performs the three main operations in a strict order, the five operations of ABSTRACTMINCOV can be executed in any order. It follows that new exploration heuristics, for instance the early discarding of subtrees after the discovering of an acceleration, can be implemented without rewriting any proof of correctness or termination.

In Section 5.1, we introduce the (implicit) *coverability trees*, the central data structure of the ABSTRACTMINCOV algorithm. In Section 5.2, we present the five operations of the ABSTRACTMINCOV algorithm. Finally, in Section 5.3 we provide some elements of our termination and correctness CoQ proofs.

5.1 Coverability Trees

We implement the (implicit) *coverability trees* in CoQ as the following inductive definition `KMTree`:

```
Inductive KMTree := | Empty
                    | Br of markingc &
                        bool &
                        {ffun transition -> KMTree}.
```

As one can see, they are nearly the same as explicit coverability trees: we just remove the sequence of accelerations that was previously part of the label of a node. The invariant properties introduced for explicit coverability trees (see the end of Section 4.1) have straightforward counterparts for the coverability trees, which are similarly maintained throughout any execution of ABSTRACTMINCOV.

5.2 The Algorithm

ABSTRACTMINCOV also consists of a main `while` loop that updates a pair (\mathcal{T}, A) , where \mathcal{T} is a coverability tree instead of an explicit one, and A a finite sequence of accelerations. Initially, the ABSTRACTMINCOV algorithm begins with the pair (\mathcal{T}, A) where A is the empty sequence ε and \mathcal{T} is the coverability tree reduced to a single valid front node N_ε labeled by $\lambda(N_\varepsilon) = x_0$. This tree is built by the CoQ function `KMTree_init`. Then, at each round of the loop, it picks one of the five operations it can apply on the pair, the one whose precondition is met, and apply it. It terminates when none of the operations have preconditions satisfied by the pair (\mathcal{T}, A) . At the end, A is discarded and only \mathcal{T} is returned. As ABSTRACTMINCOV is nondeterministic, we implement it as a relation, like we do for MINCOV. More precisely, we implement it in CoQ as a binary relation `Rel_small_step` on those pairs (\mathcal{T}, A) such that `Rel_small_step (T', A') (T, A)` corresponds to a step of ABSTRACTMINCOV from (\mathcal{T}, A) to (\mathcal{T}', A') . Hence all possible executions of ABSTRACTMINCOV

are encoded into decreasing sequences of `Rel_small_step`. Hence, by proving its well-foundedness and its correctness, we prove that every execution of the `ABSTRACTMINCOV` algorithm is correct and terminates.

Variant `Rel_small_step` :

```
(KMTree * seq acceleration) -> (KMTree * seq acceleration) -> Prop :=
| Rel_small_step_sat [...] (* saturating operation *)
| Rel_small_step_cln [...] (* cleaning operation *)
| Rel_small_step_acc [...] (* accelerating operation *)
| Rel_small_step_cov [...] (* covering operation *)
| Rel_small_step_exp [...] (* exploring operation *).
```

In the file `MinCov.v`, operations of `MINCOV` are proved to be simulated by sequences of `AbstractMinCov` operations matching the following regular expressions (for readability, the prefixes `Rel_` and `Rel_small_step_` are removed):

$$\text{clean} \subseteq \text{sat}^* \text{cln} \quad \text{accel} \subseteq \text{sat}^* \text{acc} \quad \text{explo} \subseteq \text{sat}^* \text{cov}^* \text{exp}$$

In `MINCOV`, accelerations are added to the set A only during the accelerating operation, and the added acceleration comes from the considered branch of the tree. On the contrary, the five operations of `ABSTRACTMINCOV` allow new accelerations to be added to A . Such accelerations could be computed from the tree like in `MINCOV`, but they could also be discovered by running an external heuristic algorithm for example.

The *saturating operation* is a small-step version of the already seen function `saturate_KMTree`, applying only one acceleration at a time instead of applying as many accelerations as possible. It can be performed on any front node N of label x and address `ad` such that $x \xrightarrow{a} y$ (i.e. $y = \text{apply_transitionc } x \ a$) and $x \neq y$, for some $a \in A$ and some ω -marking y . The saturating operation simply sets $\lambda(N)$ to y (which is what the function `saturate_a_little a T ad` does).

```
Rel_small_step_sat T A A' ad mc (a:acceleration) mc': Is_Front T ad
-> List.In a A
-> Some mc = m_from_add T ad
-> Some mc' = apply_transitionc mc a
-> mc != mc'
-> Rel_small_step (saturate_a_little a T ad, A'++A) (T,A)
```

The *cleaning operation* is basically the same as the one of `MINCOV`. The difference is that now the ω -marking of the considered node is required to be already saturated (which can be obtained via the `Rel_small_step_sat` operation). Also note that the `removeE_add` function has been replaced by the `remove_add` function (with the same behavior) because of the change from `KMTE` to `KMTree`. This is also the case for several other functions in the other operations.

```

Rel_small_step_cln T A A' ad: Is_Front T ad
  -> saturated_node A T ad
  -> ad_covered_not_front T ad
  -> Rel_small_step (remove_add T ad, A'++A) (T,A)

```

The *accelerating operation* is abstracted compared to the MINCOV equivalent operation. More precisely, the acceleration used to justify the cut of the branch via the `to_Front` function may come from previous stages of the algorithm, or be guessed during the operation. In the latter case, the acceleration may be computed as in MINCOV. It follows that subtrees rooted in non-saturated nodes can be discarded earlier than in MINCOV.

```

Rel_small_step_acc T A A' ad mc : ~~ Is_Front T ad
  -> Some mc = m_from_add T ad
  -> ~~ (saturated_markingc mc (A'++A))
  -> Rel_small_step (to_Front T ad, A'++A) (T,A)

```

The *covering operation* removes a node of \mathcal{T} when it is covered by a node in $\text{Front}(\mathcal{T})$. It corresponds to a part of the exploring operation of MINCOV. The non-prefix requirement is here to ensure that a front node does not trigger its own deletion.

```

Rel_small_step_cov T A A' ad mc ad' mc': Is_Front T ad
  -> Some mc = m_from_add T ad
  -> Some mc' = m_from_add T ad'
  -> mc' <= mc
  -> ~~ prefix ad' ad
  -> Rel_small_step (remove_add T ad', A'++A) (T,A)

```

The *exploring operation* is an abstracted version of the one in MINCOV. It only performs the extension of some front node N without any additional transformation. However, stronger requirements are needed. Namely, N must be already saturated (this can be obtained thanks to the saturating operation), and the non-front nodes must satisfy the `Not_Front_Antichain` property once the front flag of N is switched to `false` (this can be obtained thanks to the covering operation).

```

Rel_small_step_exp T A A' ad: Is_Front T ad
  -> saturated_node A T ad
  -> Not_Front_Antichain (remove_Front T ad)
  -> Rel_small_step (Front_extension T ad, A'++A) (T,A).

```

5.3 Certification

Termination proofs of Karp-Miller algorithms are usually based on the fact that \leq is a well-quasi-order over the set of ω -markings. As in [33], we replace this

classical notion with the notion of *almost-full* relation [32]. This order is however just an ingredient and further arguments are needed. This is especially true for MINCOV, because the tree maintained in this algorithm may not only grow, as in the original Karp-Miller algorithm, but also shrink. The code can be found in the file `Termination.v`, including the following theorem, where `Acc` is the predicate of the Coq standard library used in the constructive definition of well-foundedness.

```
Theorem wf_Rel_small_step: forall (T : KMTTree) (A : seq acceleration),
  Front_leaves T ->
  Not_Front_Antichain T ->
  Acc Rel_small_step (T,A).
```

This theorem is proved thanks to a general well-founded rewriting relation on trees described in the file `wbr_tree.v`.

Our correctness proof in COQ is close to the pen-and-paper one of MINCOV [12]. Whereas the correctness proof of the original Karp-Miller algorithm is based on branches, operations on trees performed by MINCOV depend on the complete tree. The correctness proof can be found in the file `Correctness.v`, whose main theorem is the following one, where `clos_refl_trans_1n` is the predicate for the reflexive and transitive closure, and `Markings_of_T` computes the list of all ω -markings of the input coverability tree.

```
Theorem Correctness T A (m0: marking):
  clos_refl_trans_1n _ Rel_small_step (T,A) (KMTTree_init m0) ->
  (forall T' A', ~ Rel_small_step (T',A') (T,A)) ->
  clover m0 (Markings_of_T T).
```

As in [12], this theorem is a corollary of two results, corresponding to the two directions of the equivalence in the `clover` definition.

The main theorem of the file `KMTrees.v`, shown below, provides the first direction by observing that the desired implication follows from the consistent properties mentioned in Sections 4.1 and 5.1. The fact that these properties are invariant (proved in file `AbstractMinCov.v`) implies that this implication is in fact satisfied throughout the execution and not just when the algorithm has terminated.

```
Theorem cover_consistent_KMTTree A m0 T:
  consistent_tree A T ->
  consistent_head A m0 T ->
  forall (mc: markingc) m,
  mc \in Markings_of_T T ->
  m \in mc ->
  coverable m0 m.
```

The other direction is the main theorem of file `Completeness.v`.

```

Theorem Rel_small_step_all_covered T A (m0: marking):
  clos_refl_trans_1n _ Rel_small_step (T,A) (KMTree_init m0) ->
  (forall T' A', ~ Rel_small_step (T',A') (T,A)) ->
  forall m, coverable m0 m -> exists (mc:markingc),
  mc \in Markings_of_T T /\
  m \in mc.

```

The following table summarizes the size of [33]’s and our formalizations. We import and use all files from [33] except the Karp-Miller part.

[33] (commit bbb0668)	Technical tools	631 lines
	Petri net	1226 lines
	Karp-Miller	775 lines
[This paper]	Technical tools	1790 lines
	Petri net extension	1869 lines
	MINCOV and ABSTRACTMINCOV	5590 lines

6 Conclusion

We provide a complete COQ certification of MINCOV, an algorithm that computes the minimal basis of the coverability set (of a Petri net with an initial marking). Our development is obtained by introducing a small-step variant of that algorithm, called ABSTRACTMINCOV. This variant consists of smaller and more abstract steps than in MINCOV, and which can be performed in any order. This gives a lot of freedom to an actual implementation of the algorithm, leaving room for heuristics. In particular, the step `Rel_small_step_acc` can prune *any* subtree rooted on a non-saturated node. Note that such a subtree is necessarily removed at some step of the MINCOV algorithm, since every node is saturated when the algorithm terminates. This early removal will decrease the total number of node comparisons that are performed by operations maintaining the antichain property (`Rel_small_step_cln` and `Rel_small_step_cov`). It would be interesting to quantify the actual impact of such a strategy, and more generally, of all the heuristics permitted by our ABSTRACTMINCOV algorithm.

The constructive logic of COQ provides automatic correct-by-construction OCAML code extraction. This is however not currently possible because we use relations to describe the algorithms in order to preserve their non-determinism. It should be interesting in a future work to implement choice functions and boolean versions of our **Prop** predicates, and to benchmark the extracted code against the existing PYTHON implementation of MINCOV. Since most of our predicates are already boolean functions (although their boolean natures are hidden by a coercion), we think that obtaining an OCaml extraction would be reasonably easy. However, obtaining an efficient one would require a significant additional amount of work.

Acknowledgments. We thank the anonymous reviewers for their numerous and very interesting remarks.

References

1. Angeli, D., Leenheer, P.D., Sontag, E.D.: Persistence results for chemical reaction networks with time-dependent kinetics and no global conservation laws. *SIAM Journal on Applied Mathematics* **71**(1), 128–146 (2011). <https://doi.org/10.1137/090779401>, <http://www.jstor.org/stable/41111581>
2. Baldan, P., Cocco, N., Marin, A., Simeoni, M.: Petri nets for modelling metabolic pathways: A survey. *Natural Computing* **9**, 955–989 (12 2010). <https://doi.org/10.1007/s11047-010-9180-6>
3. Blondin, M., Haase, C., Offtermatt, P.: Directed Reachability for Infinite-State Systems. In: Groote, J.F., Larsen, K.G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 12652, pp. 3–23. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_1
4. Bozzelli, L., Ganty, P.: Complexity Analysis of the Backward Coverability Algorithm for VASS. In: Delzanno, G., Potapov, I. (eds.) *Reachability Problems - 5th International Workshop, RP 2011, Genoa, Italy, September 28-30, 2011. Proceedings. Lecture Notes in Computer Science*, vol. 6945, pp. 96–109. Springer (2011). https://doi.org/10.1007/978-3-642-24288-5_10
5. Czerwinski, W., Lasota, S., Lazic, R., Leroux, J., Mazowiecki, F.: The reachability problem for Petri nets is not elementary. In: Charikar, M., Cohen, E. (eds.) *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019. pp. 24–33. ACM* (2019). <https://doi.org/10.1145/3313276.3316369>
6. Czerwinski, W., Orlikowski, L.: Reachability in Vector Addition Systems is Ackermann-complete. In: *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022. pp. 1229–1240. IEEE* (2021). <https://doi.org/10.1109/FOCS52979.2021.00120>
7. Dixon, A., Lazic, R.: KReach: A Tool for Reachability in Petri Nets. In: Biere, A., Parker, D. (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 I. Lecture Notes in Computer Science*, vol. 12078, pp. 405–412. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_22
8. Figueira, D., Figueira, S., Schmitz, S., Schnoebelen, P.: Ackermannian and Primitive-Recursive Bounds with Dickson’s Lemma. In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada. pp. 269–278. IEEE Computer Society* (2011). <https://doi.org/10.1109/LICS.2011.39>
9. Finkel, A.: The Minimal Coverability Graph for Petri Nets. In: Rozenberg, G. (ed.) *Advances in Petri Nets 1993, Papers from the 12th International Conference on Applications and Theory of Petri Nets, Gjorn, Denmark, June 1991. Lecture Notes in Computer Science*, vol. 674, pp. 210–243. Springer (1991). https://doi.org/10.1007/3-540-56689-9_45
10. Finkel, A., Geeraerts, G., Raskin, J.F., Van Begin, L.: A counter-example to the minimal coverability tree algorithm. *Université Libre de Bruxelles, Tech. Rep* **535** (2005)

11. Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, part I: completions. *Math. Struct. Comput. Sci.* **30**(7), 752–832 (2020). <https://doi.org/10.1017/S0960129520000195>
12. Finkel, A., Haddad, S., Khmelnitsky, I.: Minimal Coverability Tree Construction Made Complete and Efficient. In: Goubault-Larrecq, J., König, B. (eds.) *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*. *Lecture Notes in Computer Science*, vol. 12077, pp. 237–256. Springer (2020). https://doi.org/10.1007/978-3-030-45231-5_13
13. Geeraerts, G., Raskin, J.F., Van Begin, L.: On the Efficient Computation of the Minimal Coverability Set for Petri Nets. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) *Automated Technology for Verification and Analysis*. pp. 98–113. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75596-8_9
14. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Ph.D. thesis, Inria Saclay Ile de France (2016)
15. Hack, M.: *Decidability Questions for Petri Nets*. *Outstanding Dissertations in the Computer Sciences*, Garland Publishing, New York (1975)
16. Hilaire, T., Ilcinkas, D., Leroux, J.: Petri-net-in-coq (2024), <https://archive.softwareheritage.org/swh:1:rev:7b5523e30026266c471c73e911f0fda525c6f900;origin=https://gitub.u-bordeaux.fr/thhilaire/petri-net-in-coq.git>
17. Jančar, P.: Decidability of a Temporal Logic Problem for Petri Nets. *Theor. Comput. Sci.* **74**(1), 71–93 (1990). [https://doi.org/10.1016/0304-3975\(90\)90006-4](https://doi.org/10.1016/0304-3975(90)90006-4)
18. Kaiser, A., Kroening, D., Wahl, T.: Efficient Coverability Analysis by Proof Minimization. In: Koutny, M., Ulidowski, I. (eds.) *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4–7, 2012. Proceedings*. *Lecture Notes in Computer Science*, vol. 7454, pp. 500–515. Springer (2012). https://doi.org/10.1007/978-3-642-32940-1_35
19. Karp, R.M., Miller, R.E.: Parallel Program Schemata. *J. Comput. Syst. Sci.* **3**(2), 147–195 (1969). [https://doi.org/10.1016/S0022-0000\(69\)80011-5](https://doi.org/10.1016/S0022-0000(69)80011-5)
20. Lasota, S.: Improved Ackermannian Lower Bound for the Petri Nets Reachability Problem. In: Berenbrink, P., Monmege, B. (eds.) *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15–18, 2022, Marseille, France (Virtual Conference)*. *LIPIcs*, vol. 219, pp. 46:1–46:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.STACS.2022.46>
21. Lazic, R., Schmitz, S.: The ideal view on Rackoff’s coverability technique. *Inf. Comput.* **277**, 104582 (2021). <https://doi.org/10.1016/j.ic.2020.104582>
22. Leroux, J.: Vector addition system reachability problem: a short self-contained proof. In: Ball, T., Sagiv, M. (eds.) *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011*. pp. 307–316. ACM (2011). <https://doi.org/10.1145/1926385.1926421>
23. Leroux, J.: The Reachability Problem for Petri Nets is Not Primitive Recursive. In: *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7–10, 2022*. pp. 1241–1252. IEEE (2021). <https://doi.org/10.1109/FOCS52979.2021.00121>
24. Leroux, J., Schmitz, S.: Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In: *34th Annual ACM/IEEE Symposium on Logic*

- in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13. IEEE (2019). <https://doi.org/10.1109/LICS.2019.8785796>
25. Mayr, E.W., Meyer, A.R.: The Complexity of the Finite Containment Problem for Petri Nets. *J. ACM* **28**(3), 561–576 (1981). <https://doi.org/10.1145/322261.322271>
 26. Peleg, M., Rubin, D., Altman, R.B.: Using Petri Net Tools to Study Properties and Dynamics of Biological Systems. *Journal of the American Medical Informatics Association* **12**(2), 181–199 (03 2005). <https://doi.org/10.1197/jamia.M1637>
 27. Piipponen, A., Valmari, A.: Constructing Minimal Coverability Sets. *Fundam. Informaticae* **143**(3-4), 393–414 (2016). <https://doi.org/10.3233/FI-2016-1319>
 28. Rackoff, C.: The Covering and Boundedness Problems for Vector Addition Systems. *Theor. Comput. Sci.* **6**, 223–231 (1978). [https://doi.org/10.1016/0304-3975\(78\)90036-1](https://doi.org/10.1016/0304-3975(78)90036-1)
 29. Reynier, P.A., Servais, F.: Minimal coverability set for petri nets: Karp and miller algorithm with pruning. In: International Conference on Application and Theory of Petri Nets and Concurrency. pp. 69–88. Springer (2011). https://doi.org/10.1007/978-3-642-21834-7_5
 30. Reynier, P., Servais, F.: On the Computation of the Minimal Coverability Set of Petri Nets. In: Filiot, E., Jungers, R.M., Potapov, I. (eds.) *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 11674, pp. 164–177. Springer (2019). https://doi.org/10.1007/978-3-030-30806-3_13
 31. Schmitz, S.: The complexity of reachability in vector addition systems. *ACM SIGLOG News* **3**(1), 4–21 (2016). <https://doi.org/10.1145/2893582.2893585>
 32. Vytiniotis, D., Coquand, T., Wahlstedt, D.: Stop When You Are Almost-Full - Adventures in Constructive Termination. In: Beringer, L., Felty, A.P. (eds.) *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings. Lecture Notes in Computer Science*, vol. 7406, pp. 250–265. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_17
 33. Yamamoto, M., Sekine, S., Matsumoto, S.: Formalization of Karp-Miller tree construction on petri nets. In: Bertot, Y., Vafeiadis, V. (eds.) *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. pp. 66–78. ACM (2017). <https://doi.org/10.1145/3018610.3018626>

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

A

Abdulla, Parosh Aziz [III-276](#)
Ádám, Zsófia [III-129](#), [III-330](#), [III-371](#),
[III-412](#)
Akshay, S. [I-123](#)
Aldughaim, Mohannad [III-376](#)
Aniva, Leni [I-311](#)
Artho, Cyrille [II-3](#)
Atig, Mohamed Faouzi [III-276](#)
Avni, Guy [III-153](#)
Ayaziová, Paulína [III-341](#), [III-406](#)

B

Backes, John [I-3](#)
Badings, Thom [II-258](#)
Baier, Daniel [III-359](#)
Bajczi, Levente [III-330](#), [III-371](#), [III-412](#)
Barbosa, Haniel [I-311](#)
Barrett, Clark [I-311](#)
Basa, Eliyahu [I-123](#)
Bayless, Sam [I-3](#)
Beckert, Bernhard [I-268](#)
Bentele, Manuel [III-418](#)
Beutner, Raven [II-196](#)
Beyer, Dirk [III-129](#), [III-299](#), [III-359](#)
Blanchard, Allan [I-331](#)
Bocchi, Laura [I-207](#)
Bodden, Eric [I-229](#)
Boillot, Jérôme [III-387](#)
Bork, Alexander [II-299](#)
Bozhilov, Stanimir [III-335](#), [III-381](#)
Brauß, Franz [III-376](#)

C

Cai, Yubo [II-323](#)
Chakraborty, Debraj [II-299](#)
Chakraborty, Supratik [I-123](#), [II-175](#), [III-393](#)
Chalupa, Marek [III-353](#)
Chatterjee, Prantik [II-155](#)

Chen, Xiaohong [I-350](#)
Chen, Yean-Ru [II-363](#)
Chen, Yu-Fang [I-24](#)
Chen, Zhenbang [III-347](#)
Chien, Po-Chun [III-129](#), [III-359](#), [III-365](#)
Chocholatý, David [I-24](#), [II-130](#)
Chowdhury, Md Solimul [I-34](#)
Cimatti, Alessandro [II-44](#)
Codel, Cayden R. [I-34](#)
Cordeiro, Lucas C. [III-376](#)
Correnson, Loïc [I-331](#)
Cosler, Matthias [III-45](#)

D

D'Souza, Deepak [II-175](#)
Dacík, Tomáš [I-188](#)
Dahlsen-Jensen, Mikael Bisgaard [III-194](#)
de Pol, Jaco van [III-194](#)
Dierl, Simon [II-87](#)
Dietsch, Daniel [III-418](#)
Djoudi, Adel [I-331](#)
Dobos-Kovács, Mihály [III-371](#), [III-412](#)
Dubsloff, Clemens [III-255](#)
Duong, Hai [III-24](#)
Dwyer, Matthew B. [III-24](#)

E

Ehlers, Rüdiger [I-83](#)
Eisenbarth, Thomas [III-399](#)
Erhard, Julian [III-335](#), [III-381](#)

F

Farias, Bruno [III-376](#)
Fassbender, Dennis [II-44](#)
Fedyukovich, Grigory [II-175](#)
Feng, Nick [I-3](#)
Fiedor, Tomáš [II-130](#)
Fievet, Baptiste [III-194](#)
Fiterau-Brostean, Paul [II-87](#)

Fleury, Mathias I-311
 Fried, Dror I-123
 Furbach, Florian III-276

G

Gadelha, Mikhail R. III-376
 Galgali, Varadraj II-3
 Garcia-Contreras, Isabel I-43
 Garg, Shashwat III-276
 Griggio, Alberto II-44
 Grover, Kush II-299
 Gurfinkel, Arie I-43

H

Hahn, Christopher III-45
 Hanselmann, Michael II-44
 Hari Govind, V. K. I-43
 Hasuo, Ichiro II-279
 Havlena, Vojtěch I-24, II-130
 He, Dongjie I-229
 Heinzemann, Christian II-44
 Heizmann, Matthias III-418
 Henze, Franziska II-44
 Hermanns, Holger III-255
 Heule, Marijn J. H. I-34, I-61
 Hilaire, Thibault I-370
 Holík, Lukáš I-24, II-130
 Holter, Karoliine III-335, III-381
 Hou, Zhe II-363
 Howar, Falk II-87
 Hruška, Martin II-130
 Hu, Alan J. I-3
 Huerta y Munive, Jonathan Julián I-288
 Huisman, Marieke III-71
 Husung, Nils III-255

I

Ilcinkas, David I-370
 Iqbal, Syed M. I-3

J

Jakobsen, Anna Blume III-110
 Jankola, Marek III-359
 Jansen, Nils II-258
 Jeannin, Jean-Baptiste I-248
 Jiang, Xinyu III-418
 Jiménez-Pastor, A. II-343
 Jonáš, Martin III-90, III-406
 Jonsson, Bengt II-87

Jørgensen, Rasmus Skibdahl Melanchton
 III-110
 Jung, Jean Christoph I-167
 Junges, Sebastian II-109, II-258, II-279

K

Kabra, Aditi I-144
 Kapritsos, Manos I-248
 Karakaya, Kadiray I-229
 Karmarkar, Hrishikesh III-393
 Katoen, Joost-Pieter II-237
 Kettl, Matthias III-359
 Khalimov, Ayrat I-83
 King, Andy I-207
 Klauck, Michaela II-44
 Klauke, Jonas I-229
 Klumpp, Dominik III-418
 Köhl, Maximilian A. III-255
 Kokologiannakis, Michalis II-66
 König, Lukas II-44
 Korovin, Konstantin III-376
 Kosmatov, Nikolai I-331
 Křetínský, Jan II-299
 Kruger, Loes II-109
 Kumor, Kristián III-406
 Küperkoch, Stefan II-44
 Kwiatkowska, Marta III-3

L

Lachnitt, Hanna I-311
 Lahav, Ori III-235
 Lal, Akash II-155
 Larsen, K. G. II-343
 Laurent, Jonathan I-144
 Lee, Nian-Ze III-129, III-359, III-365
 Lemberger, Thomas III-359
 Lengál, Ondřej I-24, II-130
 Leroux, Jérôme I-370
 Li, Jianxin II-217
 Li, Xianzhiyu III-376
 Lima, Leonardo I-288
 Lin, Shang-Wei II-363
 Lingsch-Rosenfeld, Marian III-359
 Loose, Nils III-399
 Luo, Linghui I-229

M

Mächtle, Felix III-399
 Madhukar, Kumar III-393

Majumdar, Rupak II-66, III-213
 Mallik, Kaushik III-153
 Manino, Edoardo III-376
 Menezes, Rafael Sá III-376
 Mertens, Hannah II-237
 Metta, Ravindra III-393
 Micskei, Zoltán III-330
 Milanese, Marco III-387
 Miné, Antoine III-387
 Mitsch, Stefan I-144
 Mohr, Stefanie II-299
 Molnár, Vince III-371, III-412
 Monat, Raphaël III-387
 Mondok, Milán III-371, III-412
 Mozumder, Nusrat Jahan III-24
 Murgia, Maurizio I-207

N

Nayak, Satya Prakash III-173
 Neider, Daniel I-167
 Neurohr, Christian I-167
 Nötzli, Andres I-311
 Novák, Jakub III-406

O

Omar, Ayham III-45
 Osama, Muhammad II-23
 Ouadjaout, Abdelraouf III-387

P

Panagou, Dimitra I-248
 Parížek, Pavel II-3
 Parolini, Francesco III-387
 Pavlogiannis, Andreas III-110
 Petrucci, Laure III-194
 Pike, Lee I-3
 Platzer, André I-144
 Podelski, Andreas III-418
 Pogudin, Gleb II-323

Q

Qu, Daohan II-3
 Quatmann, Tim II-237

R

Reynolds, Andrew I-311
 Richter, Cedric III-353

Rodrigues, Nishant I-350
 Rogalewicz, Adam I-188
 Roşu, Grigore I-350
 Rot, Jurriaan II-109, II-279
 Roy, Subhajit II-155

S

S, Sumanth Prabhu II-175
 Saan, Simmo III-335, III-381
 Sadhukhan, Suman III-153
 Sağlam, Irmak III-213
 Sagonas, Konstantinos II-87
 Sanán, David II-363
 Sanders, Peter I-268
 Scheucher, Manfred I-61
 Schmidt, Markus I-229
 Schmitt, Frederik III-45
 Schmuck, Anne-Kathrin III-173
 Schott, Stefan I-229
 Schüssele, Frank III-418
 Schwarz, Michael III-335, III-381
 Sebe, Mircea Octavian I-350
 Sedláček, Jindřich III-406
 Seidl, Helmut III-335, III-381
 Shmarov, Fedor III-376
 Shoham, Sharon I-43
 Síč, Juraj I-24, II-130
 Sieck, Florian III-399
 Singh, Abhishek Kr III-235
 Sirrenberg, Nils III-129
 Solanki, Mayank II-155
 Somorjai, Márk III-371, III-412
 Song, Kunjian III-376
 Spiessl, Martin III-359
 Stoelinga, Marielle II-258
 Strejček, Jan III-90, III-341, III-406
 Szekeres, Dániel III-371, III-412

T

Tachna-Fram, Avi I-248
 Tåquist, Fredrik II-87
 Tekriwal, Mohit I-248
 Telbisz, Csanád III-371, III-412
 Temel, Mertcan I-340
 Teo, Yon Shin II-363
 Thejaswini, K. S. III-213
 Tihanyi, Norbert III-376

Tilscher, Sarah [III-335](#), [III-381](#)
 Tinelli, Cesare [I-311](#)
 Tomov, Naum [I-103](#)
 Tonetta, Stefano [II-44](#)
 Traytel, Dmitriy [I-288](#)
 Trentin, Patrick [I-3](#)
 Tribastone, M. [II-343](#)
 Trtík, Marek [III-90](#), [III-406](#)
 Tschaikowski, M. [II-343](#)

U

Ulbrich, Mattias [I-268](#)
 Urban, Lukáš [III-90](#)

V

Vafeiadis, Viktor [II-66](#)
 van Abbema, Feije [I-103](#)
 van de Pol, Jaco [III-110](#)
 van den Brand, Mark [III-71](#)
 van den Haak, Lars B. [III-71](#)
 van der Vegt, Marck [II-279](#)
 van Dijk, Tom [I-103](#)
 Venkatesh, R. [II-175](#), [III-393](#)
 Vojdani, Vesal [III-335](#), [III-381](#)
 Vojnar, Tomáš [I-188](#)
 Volk, Matthias [II-258](#)

W

Wachowitz, Henrik [III-359](#)
 Wang, Benjie [III-3](#)
 Wang, Tzu-Fan [II-363](#)
 Wang, Zhen [III-347](#)
 Watanabe, Kazuki [II-279](#)
 Wendler, Philipp [III-359](#)
 Westhofen, Lukas [I-167](#)
 Whalen, Mike [I-3](#)
 Wiesler, Julian [I-268](#)
 Wijs, Anton [II-23](#), [III-71](#)
 Winkler, Tobias [II-237](#)
 Witt, Sascha [I-268](#)

X

Xu, Dong [III-24](#)

Y

Yi, Pu (Luke) [II-3](#)

Z

Zaoral, Lukáš [III-406](#)
 Zhang, Leping [II-217](#)
 Zhang, Xiyue [III-3](#)
 Zhao, Yongwang [II-217](#)
 Zuleger, Florian [I-188](#)