LNCS 13243

ARCoSS

Dana Fisman
Grigore Rosu (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

28th International Conference, TACAS 2022
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022
Munich, Germany, April 2–7, 2022
Proceedings, Part I

1 Part I

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

OPEN ACCESS

Lecture Notes in Computer Science 13243

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

More information about this series at https://link.springer.com/bookseries/558

Dana Fisman · Grigore Rosu (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

28th International Conference, TACAS 2022
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2022
Munich, Germany, April 2–7, 2022
Proceedings, Part I

Springer

Editors
Dana Fisman
Ben-Gurion University of the Negev
Be’er Sheva, Israel

Grigore Rosu
University of Illinois Urbana-Champaign
Urbana, IL, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-99523-2 ISBN 978-3-030-99524-9 (eBook)
https://doi.org/10.1007/978-3-030-99524-9

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

ETAPS Foreword

Welcome to the 25th ETAPS! ETAPS 2022 took place in Munich, the beautiful capital of Bavaria, in Germany.

ETAPS 2022 is the 25th 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. Organizing these conferences in a coherent, highly synchronized conference program 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 attract many researchers from all over the globe.

ETAPS 2022 received 362 submissions in total, 111 of which were accepted, yielding an overall acceptance rate of 30.7%. 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 2022 featured the unifying invited speakers Alexandra Silva (University College London, UK, and Cornell University, USA) and Tomáš Vojnar (Brno University of Technology, Czech Republic) and the conference-specific invited speakers Nathalie Bertrand (Inria Rennes, France) for FoSSaCS and Lenore Zuck (University of Illinois at Chicago, USA) for TACAS. Invited tutorials were provided by Stacey Jeffery (CWI and QuSoft, The Netherlands) on quantum computing and Nicholas Lane (University of Cambridge and Samsung AI Lab, UK) on federated learning.

As this event was the 25th edition of ETAPS, part of the program was a special celebration where we looked back on the achievements of ETAPS and its constituting conferences in the past, but we also looked into the future, and discussed the challenges ahead for research in software science. This edition also reinstated the ETAPS mentoring workshop for PhD students.

ETAPS 2022 took place in Munich, Germany, and was organized jointly by the Technical University of Munich (TUM) and the LMU Munich. The former was founded in 1868, and the latter in 1472 as the 6th oldest German university still running today. Together, they have 100,000 enrolled students, regularly rank among the top 100 universities worldwide (with TUM's computer-science department ranked #1 in the European Union), and their researchers and alumni include 60 Nobel laureates.

The local organization team consisted of Jan Křetínský (general chair), Dirk Beyer (general, financial, and workshop chair), Julia Eisentraut (organization chair), and Alexandros Evangelidis (local proceedings chair).

ETAPS 2022 was further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology).

The ETAPS Steering Committee consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Holger Hermanns (Saarbrücken), Marieke Huisman (Twente, chair), Jan Kofroň (Prague), Barbara König (Duisburg), Thomas Noll (Aachen), Caterina Urban (Paris), Tarmo Uustalu (Reykjavik and Tallinn), and Lenore Zuck (Chicago).

Other members of the Steering Committee are Patricia Bouyer (Paris), Einar Broch Johnsen (Oslo), Dana Fisman (Be'er Sheva), Reiko Heckel (Leicester), Joost-Pieter Katoen (Aachen and Twente), Fabrice Kordon (Paris), Jan Křetínský (Munich), Orna Kupferman (Jerusalem), Leen Lambers (Cottbus), Tiziana Margaria (Limerick), Andrew M. Pitts (Cambridge), Elizabeth Polgreen (Edinburgh), Grigore Roşu (Illinois), Peter Ryan (Luxembourg), Sriram Sankaranarayanan (Boulder), Don Sannella (Edinburgh), Lutz Schröder (Erlangen), Ilya Sergey (Singapore), Natasha Sharygina (Lugano), Pawel Sobocinski (Tallinn), Peter Thiemann (Freiburg), Sebastián Uchitel (London and Buenos Aires), Jan Vitek (Prague), Andrzej Wasowski (Copenhagen), Thomas Wies (New York), Anton Wijs (Eindhoven), and Manuel Wimmer (Linz).

I'd like to take this opportunity to thank all authors, attendees, organizers of the satellite workshops, and Springer-Verlag GmbH for their support. I hope you all enjoyed ETAPS 2022.

Finally, a big thanks to Jan, Julia, Dirk, and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

February 2022

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

TACAS 2022 was the 28th edition of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022 was part of the 25th European Joint Conferences on Theory and Practice of Software (ETAPS 2022), which was held from April 2 to April 7 in Munich, Germany, as well as online due to the COVID-19 pandemic. 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 computer-controlled systems.

There were four submission categories for TACAS 2022:

1. Research papers advancing the theoretical foundations for the construction and analysis of systems.
2. Case study papers with an emphasis on a real-world setting.
3. Regular tool papers presenting a new tool, a new tool component, or novel extensions to an existing tool.
4. Tool demonstration papers focusing on the usage aspects of tools.

Papers of categories 1–3 were restricted to 16 pages, and papers of category 4 to six pages.

This year 159 papers were submitted to TACAS, consisting of 112 research papers, five case study papers, 33 regular tool papers, and nine tool demo papers. Authors were allowed to submit up to four papers. Each paper was reviewed by three Program Committee (PC) members, who made use of subreviewers. Similarly to previous years, it was possible to submit an artifact alongside a paper, which was mandatory for regular tool and tool demo papers.

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. The evaluation consisted of two rounds. The first 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 took place after paper acceptance notifications were sent out; authors of accepted research papers who did not submit an artifact in the first round could submit their artifact at this time. In total, 86 artifacts were submitted (79 in the first round and seven in the second) and evaluated by the AEC regarding their availability, functionality, and/or reusability. Papers with an artifact that was successfully evaluated include one or more badges on the first page, certifying the respective properties.

Selected authors were requested to provide a rebuttal for both papers and artifacts in case a review gave rise to questions. Using the review reports and rebuttals, the Program and the Artifact Evaluation Committees extensively discussed the papers and artifacts and ultimately decided to accept 33 research papers, one case study, 12 tool papers, and four tool demos.

This corresponds to an acceptance rate of 29.46% for research papers and an overall acceptance rate of 31.44%.

Besides the regular conference papers, this two-volume proceedings also contains 16 short papers that describe the participating verification systems and a competition report presenting the results of the 11th SV-COMP, the competition on automatic software verifiers for C and Java programs. These papers were reviewed by a separate Program Committee (PC); each of the papers was assessed by at least three reviewers. A total of 47 verification systems with developers from 11 countries entered the systematic comparative evaluation, including four submissions from industry. Two sessions in the TACAS program were reserved for the presentation of the results: (1) a summary by the competition chair and of the participating tools by the developer teams in the first session, and (2) an open community meeting in the second session.

We would like to thank all the people who helped to make TACAS 2022 successful. First, we would like to thank the authors for submitting their papers to TACAS 2022. The PC members and additional reviewers did a great job in reviewing papers: they contributed informed and detailed reports and engaged in the PC discussions. We also thank the steering committee, and especially its chair, Joost-Pieter Katoen, for his valuable advice. Lastly, we would like to thank the overall organization team of ETAPS 2022.

April 2022

Dana Fisman
Grigore Rosu
PC Chairs

Swen Jacobs
Andrew Reynolds
AEC Chairs, Tools, and Case-study Chairs

Dirk Beyer
Competition Chair

Organization

Program Committee

Parosh Aziz Abdulla	Uppsala University, Sweden
Luca Aceto	Reykjavik University, Iceland
Timos Antonopoulos	Yale University, USA
Saddek Bensalem	Verimag, France
Dirk Beyer	LMU Munich, Germany
Nikolaj Bjorner	Microsoft, USA
Jasmin Blanchette	Vrije Universiteit Amsterdam, The Netherlands
Udi Boker	Interdisciplinary Center Herzliya, Israel
Hana Chockler	King's College London, UK
Rance Cleaveland	University of Maryland, USA
Alessandro Coglio	Kestrel Institute, USA
Pedro R. D'Argenio	Universidad Nacional de Córdoba, Argentina
Javier Esparza	Technical University of Munich, Germany
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Dana Fisman (Chair)	Ben-Gurion University, Israel
Martin Fränzle	University of Oldenburg, Germany
Felipe Gorostiaga	IMDEA Software Institute, Spain
Susanne Graf	Université Joseph Fourier, France
Radu Grosu	Stony Brook University, USA
Arie Gurfinkel	University of Waterloo, Canada
Klaus Havelund	Jet Propulsion Laboratory, USA
Holger Hermanns	Saarland University, Germany
Falk Howar	TU Clausthal / IPSSE, Germany
Swen Jacobs	CISPA Helmholtz Center for Information Security, Germany
Ranjit Jhala	University of California, San Diego, USA
Jan Kretinsky	Technical University of Munich, Germany
Viktor Kuncak	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Kim Larsen	Aalborg University, Denmark
Konstantinos Mamouras	Rice University, USA
Daniel Neider	Max Planck Institute for Software Systems, Germany
Dejan Nickovic	AIT Austrian Institute of Technology, Austria
Corina Pasareanu	Carnegie Mellon University, NASA, and KBR, USA
Doron Peled	Bar Ilan University, Israel
Anna Philippou	University of Cyprus, Cyprus
Andrew Reynolds	University of Iowa, USA

Grigore Rosu (Chair)	University of Illinois at Urbana-Champaign, USA
Kristin Yvonne Rozier	Iowa State University, USA
Cesar Sanchez	IMDEA Software Institute, Spain
Sven Schewe	University of Liverpool, UK
Natasha Sharygina	Università della Svizzera italiana, Italy
Jan Strejček	Masaryk University, Czech Republic
Cesare Tinelli	University of Iowa, USA
Stavros Tripakis	Northeastern University, USA
Frits Vaandrager	Radboud University, The Netherlands
Tomas Vojnar	Brno University of Technology, Czech Republic
Christoph M. Wintersteiger	Microsoft, USA
Lijun Zhang	Institute of Software, Chinese Academy of Sciences, China
Lingming Zhang	University of Illinois at Urbana-Champaign, USA
Lenore Zuck	University of Illinois at Chicago, USA

Artifact Evaluation Committee

Pavel Andrianov	Ivannikov Institute for System Programming of the RAS, Russia
Michael Backenköhler	Saarland University, Germany
Sebastian Biewer	Saarland University, Germany
Benjamin Bisping	TU Berlin, Germany
Olav Bunte	Eindhoven University of Technology, The Netherlands
Damien Busatto-Gaston	Université Libre de Bruxelles, Belgium
Marek Chalupa	IST Austria, Austria, and Masaryk University, Czech Republic
Priyanka Darke	Tata Consultancy Services, India
Alexandre Duret-Lutz	LRDE, France
Shenghua Feng	Institute of Software, Chinese Academy of Sciences, Beijing, China
Mathias Fleury	University of Freiburg, Germany
Kush Grover	Technical University of Munich, Germany
Dominik Harmim	Brno University of Technology, Czech Republic
Swen Jacobs (Chair)	CISPA Helmholtz Center for Information Security, Germany
Xiangyu Jin	Institute of Software, Chinese Academy of Sciences
Juraj Sič	Masaryk University, Czech Republic
Daniela Kaufmann	Johannes Kepler University Linz, Austria
Maximilian Alexander Köhl	Saarland University, Germany
Mitja Kulczynski	Kiel University, Germany
Maurice Laveaux	Eindhoven University of Technology, The Netherlands
Yong Li	Institute of Software, Chinese Academy of Sciences, China
Debasmita Lohar	Max Planck Institute for Software Systems, Germany
Makai Mann	Stanford University, USA

Fabian Meyer	RWTH Aachen University, Germany
Stefanie Mohr	Technical University of Munich, Germany
Malte Mues	TU Dortmund, Germany
Yuki Nishida	Kyoto University, Japan
Philip Offtermatt	Université de Sherbrooke, Canada
Muhammad Osama	Eindhoven University of Technology, The Netherlands
Jiří Pavela	Brno University of Technology, Czech Republic
Adrien Pommellet	LRDE, France
Mathias Preiner	Stanford University, USA
José Proença	CISTER-ISEP and HASLab-INESC TEC, Portugal
Tim Quatmann	RWTH Aachen University, Germany
Etienne Renault	LRDE, France
Andrew Reynolds (Chair)	University of Iowa, USA
Mouhammad Sakr	University of Luxembourg, Luxembourg
Morten Konggaard Schou	Aalborg University, Denmark
Philipp Schlehuber-Caissier	LRDE, France
Hans-Jörg Schurr	Inria Nancy - Grand Est, France
Michael Schwarz	Technische Universität München, Germany
Joseph Scott	University of Waterloo, Canada
Ali Shamakhi	Tehran Institute for Advanced Studies, Iran
Lei Shi	University of Pennsylvania, USA
Matthew Sotoudeh	University of California, Davis, USA
Jip Spel	RWTH Aachen University, Germany
Veronika Šoková	Brno University of Technology, Czech Republic

Program Committee and Jury — SV-COMP

Fatimah Aljaafari	University of Manchester, UK
Lei Bu	Nanjing University, China
Thomas Bunk	LMU Munich, Germany
Marek Chalupa	Masaryk University, Czech Republic
Priyanka Darke	Tata Consultancy Services, India
Daniel Dietsch	University of Freiburg, Germany
Gidon Ernst	LMU Munich, Germany
Fei He	Tsinghua University, China
Matthias Heizmann	University of Freiburg, Germany
Jera Hensel	RWTH Aachen University, Germany
Falk Howar	TU Dortmund, Germany
Soha Hussein	University of Minnesota, USA
Dominik Klumpp	University of Freiburg, Germany
Henrich Lauko	Masaryk University, Czech Republic
Will Leeson	University of Virginia, USA
Xie Li	Chinese Academy of Sciences, China
Viktor Malík	Brno University of Technology, Czech Republic
Raveendra Kumar Medicherla	Tata Consultancy Services, India

Rafael Sá Menezes	University of Manchester, UK
Vince Molnár	Budapest University of Technology and Economics, Hungary
Hernán Ponce de León	Bundeswehr University Munich, Germany
Cedric Richter	University of Oldenburg, Germany
Simmo Saan	University of Tartu, Estonia
Emerson Sales	Gran Sasso Science Institute, Italy
Peter Schrammel	University of Sussex and Diffblue, UK
Frank Schüssele	University of Freiburg, Germany
Ryan Scott	Galois, USA
Ali Shamakhi	Tehran Institute for Advanced Studies, Iran
Martin Spiessl	LMU Munich, Germany
Michael Tautschnig	Queen Mary University of London, UK
Anton Vasilyev	ISP RAS, Russia
Vesal Vojdani	University of Tartu, Estonia

Steering Committee

Dirk Beyer	Ludwig-Maximilians-Universität München, Germany
Rance Cleaveland	University of Maryland, USA
Holger Hermanns	Universität des Saarlandes, Germany
Joost-Pieter Katoen (Chair)	RWTH Aachen University, Germany, and Universiteit Twente, The Netherlands
Kim G. Larsen	Aalborg University, Denmark
Bernhard Steffen	Technische Universität Dortmund, Germany

Additional Reviewers

Abraham, Erika
Aguilar, Edgar
Akshay, S.
Asadi, Sepideh
Attard, Duncan
Avni, Guy
Azeem, Muqsit
Bacci, Giorgio
Balasubramanian, A. R.
Barbanera, Franco
Bard, Joachim
Basset, Nicolas
Bendík, Jaroslav
Berani Abdelwahab, Erzana
Beutner, Raven
Bhandary, Shrajan
Biewer, Sebastian
Blicha, Martin
Brandstätter, Andreas
Bright, Curtis
Britikov, Konstantin
Brunnbauer, Axel
Capretto, Margarita
Castiglioni, Valentina
Castro, Pablo
Ceska, Milan
Chadha, Rohit
Chalupa, Marek
Changshun, Wu
Chen, Xiaohong
Cruciani, Emilio
Dahmen, Sander
Dang, Thao
Danielsson, Luis Miguel

Degiovanni, Renzo
Dell'Erba, Daniele
Demasi, Ramiro
Desharnais, Martin
Dierl, Simon
Dubslaff, Clemens
Egolf, Derek
Evangelidis, Alexandros
Fedyukovich, Grigory
Fiedor, Jan
Fitzpatrick, Stephen
Fleury, Mathias
Frenkel, Hadar
Gamboa Guzman, Laura P.
Garcia-Contreras, Isabel
Gianola, Alessandro
Goorden, Martijn
Gorostiaga, Felipe
Gorrieri, Roberto
Grahn, Samuel
Grastien, Alban
Grover, Kush
Grünbacher, Sophie
Guha, Shibashis
Gutiérrez Brida, Simón Emmanuel
Havlena, Vojtěch
He, Jie
Helfrich, Martin
Henkel, Elisabeth
Hicks, Michael
Hirschkoff, Daniel
Hofmann, Jana
Hojjat, Hossein
Holík, Lukáš
Hospodár, Michal
Huang, Chao
Hyvärinen, Antti
Inverso, Omar
Itzhaky, Shachar
Jaksic, Stefan
Jansen, David N.
Jin, Xiangyu
Jonas, Martin
Kanav, Sudeep
Karra, Shyam Lal
Katsaros, Panagiotis
Kempa, Brian
Klauck, Michaela
Kreitz, Christoph
Kröger, Paul
Köhl, Maximilian Alexander
König, Barbara
Lahijanian, Morteza
Larraz, Daniel
Le, Nham
Lemberger, Thomas
Lengal, Ondrej
Li, Chunxiao
Li, Jianlin
Lorber, Florian
Lung, David
Luppen, Zachary
Lybech, Stian
Major, Juraj
Manganini, Giorgio
McCarthy, Eric
Mediouni, Braham Lotfi
Meggendorfer, Tobias
Meira-Goes, Romulo
Melcer, Daniel
Metzger, Niklas
Milovancevic, Dragana
Mohr, Stefanie
Najib, Muhammad
Noetzli, Andres
Nouri, Ayoub
Offtermatt, Philip
Otoni, Rodrigo
Paoletti, Nicola
Parizek, Pavel
Parker, Dave
Parys, Paweł
Passing, Noemi
Perez Dominguez, Ivan
Perez, Guillermo
Pinna, G. Michele
Pous, Damien
Priya, Siddharth
Putruele, Luciano
Pérez, Jorge A.
Qu, Meixun
Raskin, Mikhail

Rauh, Andreas
Reger, Giles
Reynouard, Raphaël
Riener, Heinz
Rogalewicz, Adam
Roy, Rajarshi
Ruemmer, Philipp
Ruijters, Enno
Schilling, Christian
Schmitt, Frederik
Schneider, Tibor
Scholl, Christoph
Schultz, William
Schupp, Stefan
Schurr, Hans-Jörg
Schwammberger, Maike
Shafiei, Nastaran
Siber, Julian
Sickert, Salomon
Singh, Gagandeep
Smith, Douglas
Somenzi, Fabio
Stewing, Richard
Stock, Gregory
Su, Yusen
Tang, Qiyi
Tibo, Alessandro
Trefler, Richard
Trtík, Marek
Turrini, Andrea
Vaezipoor, Pashootan
van Dijk, Tom
Vašíček, Ondřej
Vediramana Krishnan, Hari Govind
Wang, Wenxi
Wendler, Philipp
Westfold, Stephen
Winter, Stefan
Wolovick, Nicolás
Yakusheva, Sophia
Yang, Pengfei
Zeljić, Aleksandar
Zhou, Yuhao
Zimmermann, Martin

Contents – Part I

Synthesis

Verification

Blockchain

Grammatical Inference

Verification Inference

Short papers

Constraint Solving

Model Checking and Verification

Contents – Part II

Probabilistic Systems

Omega Automata

Equivalence Checking

Monitoring and Analysis

Tools | Optimizations, Repair and Explainability

11th Competition on Software Verification: SV-COMP 2022

Synthesis

HOLL: Program Synthesis for Higher Order Logic Locking

Gourav Takhar[1](✉), Ramesh Karri[2], Christian Pilato[3], and Subhajit Roy[1]

[1] Indian Institute of Technology Kanpur, Kanpur, India.
{tgourav,subhajit}@cse.iitk.ac.in
[2] New York University, New York, NY, USA. rkarri@nyu.edu
[3] Politecnico di Milano, Milan, Italy. christian.pilato@polimi.it

Abstract. Logic locking “hides" the functionality of a digital circuit to protect it from counterfeiting, piracy, and malicious design modifications. The original design is transformed into a “locked" design such that the circuit reveals its correct functionality only when it is “unlocked" with a *secret* sequence of bits—the *key* bit-string. However, strong attacks, especially the *SAT attack* that uses a SAT solver to recover the key bit-string, have been profoundly effective at breaking the locked circuit and recovering the circuit functionality.

We lift logic locking to *Higher Order Logic Locking (HOLL)* by hiding a higher-order *relation*, instead of a key of independent values, challenging the attacker to discover this *key relation* to recreate the circuit functionality. Our technique uses *program synthesis* to construct the locked design and synthesize a corresponding *key relation*. HOLL has low overhead and existing attacks for logic locking do not apply as the entity to be recovered is no more a value. To evaluate our proposal, we propose a new attack (*SynthAttack*) that uses an inductive synthesis algorithm guided by an operational circuit as an input-output oracle to recover the hidden functionality. SynthAttack is inspired by the SAT attack, and similar to the SAT attack, it is *verifiably correct*, i.e., if the correct functionality is revealed, a verification check guarantees the same. Our empirical analysis shows that SynthAttack can break HOLL for small circuits and small key relations, but it is ineffective for real-life designs.

Keywords: Logic Locking · Program Synthesis · Hardware Security.

1 Introduction

High manufacturing costs in advanced technology nodes are pushing many semiconductor design houses to outsource the fabrication of integrated circuits (IC) to third-party foundries [26, 42]. A *fab-less* design house can increase the investments in the chip's intellectual property, while a single foundry can serve multiple companies. However, this globalization process introduces security threats in the supply chain [25]. A malicious employee of the foundry can access and reverse engineer the circuit design to make illegal copies. Logic locking [44] alters the

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 3–24, 2022.
https://doi.org/10.1007/978-3-030-99524-9_1

chip's functionality to make it unusable by the foundry. This alteration depends on a *locking key* that is re-inserted into the chip in a trusted facility, after fabrication. The locking key is, thus, the "secret", known only to the design house. Logic locking assumes that the attackers have no access to the key but they may have access to a functioning chip (obtained, for example, from the legal/illegal market). However, logic locking has witnessed several attacks that analyze the circuit and attempt key recovery [31, 43, 48, 59].

In this paper[4], we combine the intuitions from logic locking, program synthesis, and programmable devices to design a new locking mechanism. Our technique, called *higher order logic locking* (HOLL), locks a design with a *key relation* instead of a sequence of independent *key bits*. HOLL uses *program synthesis* [3, 50] to translate the original design into a locked one. Our experiments demonstrate that HOLL is fast, scalable, and robust against attacks. Prior attacks on logic locking, like the SAT attack [51], are not practical for HOLL. Since the functionality of the key-relation is completely missing, attackers cannot simply make propositional logic queries to recover the key (like [43, 51, 18]). There are variants of logic locking, like TTLock [61] and SFLL [60], that attempt to combat SAT attacks [51]. However, these techniques use additional logic blocks (comparison and restoration circuits) which makes them prone to attacks via structural and functional analysis on this additional circuitry [47]. HOLL is resilient to such techniques as it exposes only a programmable logic that does not leak any information related to the actual functionality to be implemented.

In contrast to logic locking, attacking HOLL requires solving a second-order problem (see §8 for a detailed discussion on this). To assess the security of our method, we design a new attack, *SynthAttack*, by combining ideas from SAT attack [51] and inductive program synthesis [50]. SynthAttack employs a counterexample guided inductive synthesis (CEGIS) procedure guided via a functioning instance of the circuit as an input-output oracle. This attack constructs a synthesis query to discover key relations that invoke *semantically distinct functional behaviors* of the locked design—witnesses to such relations, referred to as *distinguishing inputs*, act as counterexamples to drive inductive learning. When the locked design is verified to have unique functionality, the attack is declared successful, with the corresponding provably-correct key relation.

Our experimental results (§6) show that the time required by an attacker to recover the key relation for a given set of distinguishing inputs (*attack time*) increases exponentially with the size of key relation. While the attacker may be able to recover key relations for small HOLL-locked circuits with small key relations, larger circuits are robust to SynthAttack. For example, for the `des` benchmark, the asymmetry between HOLL defense and SynthAttack is large; while HOLL can lock this design in less than `100` seconds, the attack cannot recover the design even within four days for a key relation that increases the area overhead of the IC by only `1.2`%. Further, the attack time required to unlock the designs increase exponentially with the complexity of the key relation.

The key relation can be implemented with reconfigurable or programmable

[4] An extended version [53] of this paper is also available.

$$
\begin{aligned}
t_0 &= x_0 \wedge x_2; \\
t_1 &= \boxed{x_3 \wedge t_0} \\
t_2 &= \boxed{(x_1 \wedge t_0)} \\
y_0 &= x_0 \oplus x_2 \\
y_2 &= (x_1 \wedge x_3) \vee t_2 \vee t_1 \\
y_1 &= t_0 \oplus x_1 \oplus x_3
\end{aligned}
$$

(a) Original circuit

$$
\begin{aligned}
t_0 &= x_0 \wedge x_2 \\
t_1 &= \boxed{(x_0 \wedge (r_4 \oplus r_2) \wedge x_3)} \\
t_2 &= \boxed{(x_0 \wedge r_3)} \\
y_0 &= x_0 \oplus x_2 \\
y_2 &= (x_1 \wedge x_3) \vee t_2 \vee t_1 \\
y_1 &= t_0 \oplus x_1 \oplus x_3
\end{aligned}
$$

(b) Locked circuit

$$
\begin{aligned}
&\{(r_0 \leftrightarrow x_1), \\
&(r_1 \leftrightarrow x_2), \\
&(r_2 \leftrightarrow \mathit{rand}), \\
&(r_3 \leftrightarrow r_0 \wedge r_1), \\
&(r_4 \leftrightarrow r_1 \oplus r_2)\}
\end{aligned}
$$

(c) Key relation

Fig. 1: HOLL on a 2-bit Adder.

devices, like programmable array logic (PAL) or embedded finite-programmable gate array (eFPGA). For example, eFPGA, essentially an IP core embedded into an ASIC or SoC, is becoming common in modern SoCs [2] and has been shown to have high resilience against bit-stream recovery [7].
Our contributions are:

- We propose a novel IP protection strategy, called *higher order logic locking* (HOLL), that uses program synthesis;
- To evaluate the security offered by HOLL, we propose a strong adversarial attack algorithm, SynthAttack;
- We build tools to apply HOLL and SynthAttack to combinational logic;
- We evaluate HOLL on cost, scalability, and robustness;

2 HOLL Overview

2.1 Threat Model: the Untrusted Foundry

We focus on the threat model where the attacker is in the foundry [44, 45] to which a fab-less design house has outsourced its IC fabrication. Such an attacker has access to the IC design and the (locked) GDSII netlist which can be reverse-engineered. Also, if the attacker can access a working IC (e.g., by procuring an IC from the open market or a discarded IC from the gray market), they can leverage the functional IC's I/O behavior as a black-box oracle. However, we assume the attacker *cannot* extract the *bitstream*, i.e. the correct sequence of configuration bits, from the device. This can be achieved with encryption techniques when the bitstream is not loaded into the device. Also, anti-readback solutions can prevent the attacker from reading the bitstream from the device. The parameters used to synthesize the *key relation* and the *locked circuit* (like the domain-specific language (DSL), budget etc.) are not known to the attacker (see §8).

2.2 Defending with *HOLL*

Consider a hardware circuit $Y \leftrightarrow \varphi(X)$, where X and Y are the inputs and outputs, respectively. HOLL uses a higher-order lock—a *secret* relation (ψ) among a certain number of additional *relation* bits R. We refer to ψ as the **key relation**.

Fig. 1a shows an example of a 2-bit adder with input X ($\{x_1x_0, x_3x_2\}$) and output Y ($y_2y_1y_0$). The circuit is locked by transforming the original expressions (marked in blue) in Fig. 1a to the *locked* expressions (marked in red) in Fig. 1b. The locked expressions use the additional *relation* bits r_2, r_3, and r_4, enabling that this *locked design* $\hat{\varphi}(X, R)$ functions correctly when the secret relation ψ (Fig. 1c) is installed. The relation ψ establishes the correct relation between the **relation bits** R. The key relation can be excited by circuit inputs (like in r_0 and r_1,), constants, or random bits (e.g., from system noise, etc.); for example, the value *rand* in Fig. 1c represents the random generation of a bit (0 or 1) assigned to r_2. The "output" from the key relation are bits r_3 and r_4 that must satisfy the relational constraints enforced by the key relation.

For the sake of simplicity, in the rest of the paper, we assume the relation bits are drawn only from the inputs X of the design. We will attempt to infer key relations of the form $\psi(X, R)$. The reader may assume the value *rand* of in Fig. 1c to be a constant value (say 0) to ease the exposition.

As $\hat{\varphi}$ also consumes the *relation bits* R, HOLL transforms the original circuit $Y \leftrightarrow \varphi(X)$ into a *locked* circuit $Y \leftrightarrow \hat{\varphi}(X, R)$ such that the locked circuit functions correctly if the key relation $\psi(X, R)$ is satisfied. In other words, HOLL is required to preserve the semantic equivalence between the original and locked designs ($\varphi = \hat{\varphi} \wedge \psi$). Note that it only imposes constraints on the input-output functionality of the circuits, not on the generated values of internal gates. For example, in Fig. 1b, the value of t_1 may be different from the one in the original design (Fig. 1a), but the final output y_2 is equivalent to the original adder.

This approach has analogies with the well-known *logic locking* solution [10, 37, 54]. Traditional logic locking produces a locked circuit by mutating certain expressions based on input key bits. HOLL differs from logic locking on the type of entities employed as hidden keys. While logic locking uses a *key value* (i.e., a sequence of key-bits), our technique uses a *key relation* (i.e., a functional relation among the key bits). As we attempt to hide a higher-order entity (relation), we refer to our scheme as *higher-order logic locking* (HOLL). As synthesizing a relation (a second-order problem) is more challenging to recover than a bit-sequence (a first-order problem), HOLL is, at least in theory, is more secure than logic locking. Our experimental results (§6) show that this security also translates to practice.

Hardware constraints. Since the key relation must be implemented in the circuit, we need to consider practical constraints. For example, the size of the key relation affects the size of the programmable logic to be used for its implementation. This, in turn, introduces area and delay overheads in the final circuit. The practical realizability of this technique adds certain hardware constraints:

- The key relation must be small for it to have a small area overhead;
- The key relation must only be executed once to avoid a significant performance overhead;
- The key relation must encode non-trivial relations between the challenge and response bits to strong security;
- The locked expressions are evenly distributed across the design to protect all parts of the circuit, disallowing focused attacks by an attacker on a small part of the circuit that contains all locks.

Inferring the key relation. HOLL operates by

1. carefully selecting a set of expressions, $E \subseteq \varphi$, in the original design φ;
2. mutating each expression $e_i \in E$ using the relation bits R to create the corresponding *locked expression*, $\hat{e}_i$.

For example, in Fig. 1a, we select two expressions, $E = \{e_1, e_2\}$ where $e_1 = x_1 \wedge t_0$ and $e_2 = x_3 \wedge t_0$. e_1 computes t_2 and is a function of t_0 and x_1, while $\hat{e}_1$ uses x_0 and r_3, which is in turn a relation of r_0 and r_1. We formalize our lock program synthesis problem as follows.

Lock Inference. Given a circuit $Y \leftrightarrow \varphi(X)$, construct a *locked circuit* $Y \leftrightarrow \hat{\varphi}(X, R)$ and a *key relation* $\psi(X, R)$ such that $\hat{\varphi}$ is *semantically equivalent* to φ with the correct relation ψ. Specifically, it requires us to construct: (1) a key relation ψ and (2) a set of locked expressions $\hat{E}$ relating to the set of selected expressions E extracted from φ such that the following conditions are met:

- **Correctness**: The circuit is guaranteed to work correctly for all inputs when the key relation is installed:

$$\forall X.\ (\forall R.\ \psi(X, R) \implies (\hat{\varphi}(X, R) = \varphi(X))) \tag{1}$$

  where $\hat{\varphi} \equiv \varphi[\hat{e}_1/e_1, \ldots, \hat{e}_n/e_n]$, for $e_i \in E \subseteq \varphi$. The notation $\varphi[e_a/e_b]$ implies that e_b is replaced by e_a in the formula φ.
- **Security**: There must exist some relation ψ' (where $\psi' \neq \psi$) where the circuit exhibits incorrect behavior; in other words, it enforces the key relation to be non-trivial:

$$\exists \psi\ \exists X\ \exists R.\ (\psi'(X, R) \implies \hat{\varphi}(X, R) \neq \varphi(X)) \tag{2}$$

We pose the above as a *program synthesis* [40, 49] problem. In particular, we search for "mutations" $\hat{e}_1, \ldots, \hat{e}_2$ and a suitable key relation ψ such that the above constraints are satisfied.

2.3 Attacking with *SynthAttack*

As we attempt to hide a relation instead of a key-value, prior attacks on logic locking (like SAT attacks), which attempt to infer key bit-strings, do not apply. However, the attackers can also use program synthesis techniques to recover the key relation using an activated instance of the circuit as an input-output oracle.

We design an attack algorithm, called *SynthAttack*, combining ideas from SAT attack (for logic locking) and counterexample guided inductive program synthesis. Our attack algorithm generates inputs $X_1, X_2, \ldots, X_n$ and computes the corresponding outputs $Y_1, Y_2, \ldots, Y_n$ using the oracle, to construct a set of *examples* $\Lambda = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$. Then, the attacker can generate a key relation ψ that satisfies the above examples, λ, using a program synthesis query:

$$\prod_{X_i, Y_i \in \Lambda} \exists R_i.\ \hat{\varphi}(X_i, R_i) \wedge \psi(X_i, R_i) = Y_i \tag{3}$$

The above query requires *copies* of $\hat{\varphi}(X, R)$ for every example—hence, the formula will quickly explode with an increasing number of samples. Our scheme is robust since the sample complexity of the key relationships increases exponentially with the number of relation bits employed. Additionally, the attacker does not know which input bits excite the key relation and how the relation bits are related to each other.

For the locked adder (Fig. 1b) with the input samples shown in Table 1 (first four rows), the above attack can synthesize the key relation shown in Fig. 2. Columns Y and $\hat{Y}$ in Table 1 represent the outputs of the original circuit and the circuit obtained by the attacker, respectively. Even on a 4-bit input space, when 25% of all possible samples are provided, the attack fails to recover the key relation as shown by the last input row of Table 1. The red box highlights the output in the attacker circuit does not match the original design.

Table 1: In-out samples.

X	Y	$\hat{Y}$
1111	110	110
1001	011	011
0000	000	000
1100	011	011
0101	010	110

$$\{(r_0 \leftrightarrow x_2),\ (r_1 \leftrightarrow x_0),\ (r_2 \leftrightarrow 0),\ (r_3 \leftrightarrow r_0 \wedge r_1),\ (r_4 \leftrightarrow 0)\}$$

Fig. 2: Generated key relation

Definition (Distinguishing Input). Given a locked circuit $\hat{\varphi}$, we refer to input X as a distinguishing input if there exist candidate relations ψ_1 and ψ_2 that evoke semantically distinct behaviors on the input X. Formally, X is a distinguishing input provided the following formula is satisfiable[5] on some relations ψ_1 and ψ_2:

$$\hat{\varphi}(X, R_1) \neq \hat{\varphi}(X, R_2) \wedge \psi_1(X, R_1) \wedge \psi_2(X, R_2) \tag{4}$$

It searches for a *distinguishing input*, X_d, that produces conflicting outputs on the locked design. Any such distinguishing input is added to the set of examples, Λ, and the query repeated. If the query is unsatisfiable, it implies that no other relation can produce a different behavior on the locked design and so the relation that satisfies the current set of examples must be a correct key relation.

Though SynthAttack significantly reduces the sample complexity of the attack, our experiments demonstrate that SynthAttack is still unsuccessful at breaking HOLL for larger designs.

3 Program Synthesis to Infer Key Relations

We represent the key relation ψ as a propositional formula, represented as a set (conjunction) of propositional terms. The terms in ψ belong are categorised as:

- **Stimulus terms**: As mentioned in §2, a subset of the relation bits are related to input bits or constants; the stimulus terms appear as $(r_i \leftarrow x_j)$ where $r_i \in R, x_j \in X \cup \{0, 1\}$.
- **Latent terms**: These clauses establish a relation among the relation bits; the variables v in these terms are drawn from the relation bits R, i.e. $v \in R$.

[5] All free variables are existentially quantified.

For example, in Fig. 1c the terms $(r_0 \leftarrow x_1)$, $(r_1 \leftarrow x_2)$, and $(r_2 \leftarrow rand)$ are stimulus terms, while $(r_3 \leftarrow (r_0 \wedge r_1))$ and $(r_4 \leftarrow (r_1 \oplus r_2))$ are latent terms.

Budget. As the key relation may need to be implemented within a limited hardware budget, our synthesis imposes a hard threshold on its size. The threshold could directly capture the hardware constraints for implementing the key relation (e.g., the estimated number of gates or ports) or indirectly indicate the complexity of the key relation (e.g., number of relation bits, propositional terms, or latent terms).

3.1 Lock and Key Inference

Algorithm 1 outlines our algorithm for inferring the key relation and the locked circuit. The algorithm accepts an unlocked design $Y \leftrightarrow \varphi(X)$ and a budget T for the key relation.

Algorithm 1: HOLL(φ, T, Q)

```
1  $\psi \leftarrow \emptyset$
2  $\hat{\varphi} \leftarrow \varphi$
3  $done \leftarrow$ False
4  while not $done$ do
5      $E \leftarrow SelectExpr(\hat{\varphi})$
6      $H, \hat{E} \leftarrow Synthesize(\psi, \hat{\varphi}, E)$
7      $\psi' \leftarrow \psi \cup H$
8      if $Budget(\psi') \leq T$ then
9          $\psi \leftarrow \psi'$
10         $\hat{\varphi} \leftarrow \hat{\varphi}[\{\hat{e}_i / e_i \mid$
11             $e_i \in E_i, \hat{e}_i \in \hat{E}\}]$
12     else
13         $q \leftarrow CheckSec(\psi, \hat{\varphi})$
14         if $q$ then
15             $done \leftarrow$ True
16         else
17             $\psi \leftarrow \emptyset$
18             $\hat{\varphi} \leftarrow \varphi$
19         end
20     end
21 end
22 return $\hat{\varphi}, \psi$
```

Main Algorithm. The algorithm iterates, increasing the complexity of the key relation, till the budget T is reached (Lines 4-21). In every iteration, the algorithm selects a set of suitable expressions E for locking, uses our synthesis procedure to extract a set of additional latent terms H for the key relation, and produces the mutated expressions $\hat{e}_i$ for each expression $e_i \in E$ (Line 6). If the additional synthesized relations keep the key relation within the budget T (Line 8), the mutated expressions are replaced for $e_i \in E$ (Line 11). HOLL verifies that the solution meets the two objectives of correctness and security (§2). We handle correctness in the *Synthesize* procedure of Algorithm 1 and security in Lines 13-14 of the same algorithm. The *CheckSec*() procedure verifies if the synthesized (locked) circuit and key relations satisfy the security condition (Eqn 2). If *CheckSec*() returns `True`, the key relation ψ and the locked circuit $\hat{\varphi}$ are returned; otherwise, synthesis is reattempted.

Correctness. HOLL attempts to synthesize (via the *Synthesize* procedure) a key relation ψ and a set of locked expressions $\hat{e}_i$ such that the circuit is guaranteed to work correctly for all inputs given to ψ; this requires us to satisfy:

$$\exists \psi, \hat{e}_1, \ldots, \hat{e}_n.\ \forall X.\ \forall R.\ (\psi(X, R) \implies \hat{\varphi}(X, R) = \varphi(X)) \quad (5)$$

$$\begin{aligned}&(r_0 \leftarrow x_1),\\ &(r_1 \leftarrow x_2),\ (r_2 \leftarrow x_0),\\ &(r_5 \leftarrow (r_0 \wedge r_1)),\\ &(r_4 \leftarrow (\boxed{(r_0 \wedge r_1)} \wedge r_2)),\\ &(r_3 \leftarrow (\boxed{(r_0 \wedge r_1)} \vee r_2))\end{aligned}$$

(a) Without optimization

$$\begin{aligned}&(r_0 \leftarrow x_1),\\ &(r_1 \leftarrow x_2),\\ &(r_2 \leftarrow x_0),\\ &(r_5 \leftarrow (r_0 \wedge r_1)),\\ &(r_3 \leftarrow (\boxed{r_5} \vee r_2)),\\ &(r_4 \leftarrow (\boxed{r_5} \wedge r_2))\end{aligned}$$

(b) With opt.

Fig. 4: Key relations generated without and with optimization.

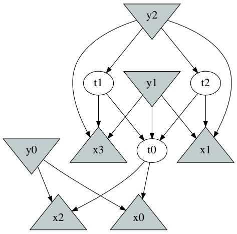


Fig. 5: Dependency graph for the expressions in Fig. 1a.

where $\hat{\varphi} \equiv \varphi[\hat{e}_1/e_1, \ldots, \hat{e}_n/e_n]$, for $e_i \in (E \subsetneq \varphi)$. In other words, we attempt to synthesize a set of modified expressions $\hat{E}$ that, once replaced the selected expressions in E, produces a semantically equivalent circuit as the original circuit if the relation ψ holds.

We solve this synthesis problem via counterexample-guided inductive synthesis (CEGIS) [3]. We provide a domain-specific language (DSL) in which ψ and $\hat{e}_i$ are synthesized. CEGIS generates candidate solutions for the synthesis problem and uses violations to the specification (i.e. the above constraint) to guide the search for suitable *programs* ψ and $\hat{e}_i$.

A problem with the above formulation is illustrated in Fig. 4: the key relation in Fig. 3a uses 5 gates without reusing expressions, "wasting" hardware resources. Fig. 3b shows an optimized key relation that reuses the response bit r_5, allowing an implementation with only 3 gates. To encourage subexpression reuse, we solve the following optimization problem where $\hat{\varphi} \equiv \varphi[\hat{e}_1/e_1, \ldots, \hat{e}_n/e_n]$, for $e_i \in E \subseteq \varphi$.:

$$\underset{\mathsf{budget}(\psi)}{\mathsf{argmin}}\ \exists \psi, \hat{e}_1, \ldots, \hat{e}_n.\ \forall X.\ (\forall R.\ \psi(X,R) \implies \hat{\varphi}(X,R) = \varphi(X)) \quad (6)$$

Security. The security objective requires that the locking (i.e., the key relation ψ and the locked expressions) is non-trivial; that is. there exists some relation ψ':$\psi' \neq \psi$ for which the circuit is not semantically equivalent to the original design:

$$\exists \psi', \psi' \neq \psi,\ \text{s.t.}\ \exists X.\ (\exists R.\ \psi'(X,R) \wedge \hat{\varphi}(X,R) \neq \varphi(X)) \quad (7)$$

The above constraint is difficult to establish while synthesizing ψ; it requires a search for a different relation ψ' that makes $\hat{\varphi}$ semantically distinct from φ while ψ maintains semantic equivalence. Instead, we use a two-pronged strategy:

- We carefully design the DSL used to synthesize ψ and $\hat{e}_i$ to reduce the possibility they generate trivial relations;
- After obtaining ψ and $\hat{\varphi}$, we attempt to synthesize an alternative relation ψ' (using 8) such that $\hat{\varphi}$ is not semantically equivalent to φ, ensuring that ψ and $\hat{\varphi}$ do not constitute a trivial locked circuit.

$$\exists \psi'.\ \exists X, R'.\ \varphi(X) \neq \hat{\varphi}(X, R') \wedge \psi'(X, R') \tag{8}$$

The procedure *CheckSec*$(\psi, \hat{\varphi})$ (Algorithm 1, Line 13) implements the above check (Eqn. 8).

Theorem 1. *If Algorithm 1 terminates, it returns a correct (Eqn. 1) and secure (Eqn. 2) locked design.*

Proof. The proof follows trivially from the design of the *Synthesize* (in particular, Eqn. 5) and *CheckSec* (in particular, Eqn. 8) procedures.

3.2 Expression Selection

HOLL constructs the dependency graph [19] (V, D) for expression selection, where nodes V are circuit variables. A node $v \in V$ represents an expression e such that v is assigned the result of expression e, i.e. $(v \leftarrow e)$. The edges D are dependencies: the edge $v_1 \rightarrow v_2$ connects the two nodes v_1 to v_2 if variable v_1 depends on variable v_2. The tree is rooted at the output variables and the input variables appear as leaves.

For example, Fig. 5 shows the dependency graph for the circuit in Fig. 1a. Triangles represent input ports (x_0, x_1, x_2, x_3) while inverted triangles represent output ports (y_0, y_1, y_2).

Our variable selection algorithm has the following goals:

1. **Ensure expression complexity**: The algorithm selects an expression e_z as a candidate for locking only if the depth of the corresponding variable z in the dependency graph lies in a user-defined range [L, U] to create a candidate set Z. The lower threshold L assures the expression captures a reasonably complex relation over the inputs, while the upper threshold U ensures the relation is not too complex to exceed the hardware budget. The algorithm starts by randomly selecting a variable $z_0 \in Z$ from this set.
2. **Encourage sub-expression reuse in key relation**: We attempt to select multiple “close" expressions; for the purpose, the algorithm randomly selects variables $w_i \subseteq Z$ on which z_0 (transitively) depend on.
3. **Encourage coverage**: We select expressions for locking in a manner so as to *cover* the circuit. To this end, interpreting (V, D) as an undirected graph, we randomly select expressions $u_i \in Z$ that are *farthest* from z_0, i.e. the shortest distance between u_i and z_0 is maximized.

Our algorithm first executes step (1), and then, indeterminately alternates between (2) and (3), till the required number of variables are selected. Let us use the dependency graph in Fig. 5 to show how the above algorithm operates:

- Given the user-defined range `[1,3]`, we compose the initial candidate set $Z = \{y_0, t_0, t_1, t_2, y_1, y_2\}$.
- Let us assume we randomly pick the expression for y_2. Now, y_2 depends on expressions t_0, t_1 and t_2 ($\{t_0, t_1, y_2\} \subseteq Z$) [*Rule 1*].

- We randomly choose new expressions to lock/transform from $\{t_0, t_1, y_2\}$. For example, we select t_2 and t_0—[*Rule 2*].
- We find y_0, which is the furthest expression from t_0, t_2, y_2 in Z. We select to lock the set of expressions $\{y_1, y_2, t_0, t_2\}$—[*Rule 3*].

4 HOLL: Implementation and Optimization

Implementation. We implemented HOLL in Python, using Sketch [49] synthesis engine to solve the program synthesis queries. We used Berkeley-ABC [8] to convert the benchmarks into Verilog and PyVerilog [52], a Python-based library, to parse the Verilog code and generate input for Sketch. We use the support for optimizing over a synthesis queries provided by Sketch to solve Eqn. 6. Algorithm 1 may not terminate; our implementation uses a timeout to ensure termination.

Domain Specific Language. We specify our domain-specific language for synthesizing our key relations and locked expressions. The grammar is specified as generators in the Sketch [49] language. The grammar for the key relations and locked expressions is as follows:

$$\begin{aligned}\langle G\rangle &::= r \leftarrow x \mid r \leftarrow r\langle Bop\rangle r \mid r \leftarrow \langle Uop\rangle r \mid r \leftarrow r\\ \langle Bop\rangle &::= or \mid and \mid xor\\ \langle Uop\rangle &::= not\end{aligned}$$

The rule $\langle G\rangle ::= r \leftarrow x$ is only present in the key relation grammar since the locked expressions have no input bits.

Backslicing. To improve scalability, we use backslicing [55] to extract the portion of the design related to the expressions selected for locking. For a variable v_i, the set of all transitive dependencies that can affect the value of v_i is referred to as its *backslice*. For example, in Fig. 5, $backslice(t2) = \{t_0, x_0, x_1, x_2\}$.

Given the set of expressions E, we compute the union of the backslices of the variables in E, i.e. all expressions B in the subgraph induced by $e \in E$ in the dependency graph; we use $B \subseteq E$ for lock synthesis.

Backslicing tilts the asymmetrical advantage further towards the HOLL defense. The attacker cannot apply backslicing on the locked design since the dependencies are obscured, preventing the extraction of the dependency graph.

Incremental Synthesis. Given a set of expressions E, the procedure *Synthesis* in Algorithm 1 creates a list of relations H and a new set of locked expressions $\hat{E}$. If the list of expressions is large, we select the expressions in the increasing order of their depth in the dependency graph. The lower the depth of the expression is, the closer it is to the inputs, and the simpler is the expression. Selecting an expression with the lowest depth first (say e_1) ensures that other expressions (e_j)

depending on this expression can use the relations H generated during synthesis of $\hat{e}_1$. This also makes synthesizing the other expressions easier as the current relation has some sub-expressions on which the new relations can be built.

5 SynthAttack: Attacking HOLL with Program Synthesis

As HOLL requires inference of relations and not values, existing attacks designed for logic locking do not apply. We design a new attack, SynthAttack, that is inspired by the SAT attack [51] for logic locking and counterexample-guided inductive program synthesis (CEGIS) [50].

5.1 The SynthAttack Algorithm

```
Algorithm 2: SynthAttack($\hat{\varphi}$, $[\![\hat{\varphi}(\psi)]\!]$)
$i \leftarrow 0$
$Q_0 \leftarrow \top$
while True do
    $X' \leftarrow Solve_X(Qi$
        $\wedge\ (\hat{\varphi}(X, R_1) \neq \hat{\varphi}(X, R_2))$
        $\wedge\ \psi_1(X, R_1) \wedge \psi_2(X, R_2))$
    if $X' = \bot$ then
        break
    end
    $Y' \leftarrow [\![\hat{\varphi}(\psi)]\!](X')$
    $Q_{i+1} \leftarrow Q_i\ \wedge\ (\hat{\varphi}(X', R_1^i) \leftrightarrow Y')$
        $\wedge\ (\hat{\varphi}(X', R_2^i) \leftrightarrow Y')$
        $\wedge\ \psi_1(X', R_1^i)\ \wedge\ \psi_2(X', R_2^i)$
    $i \leftarrow i + 1$
end
$\psi_1, \psi_2 \leftarrow Solve_{\psi_1,\psi_2}(Q_i)$
return $\psi_1$
```

SynthAttack runs a CEGIS loop: it accumulates a set of examples, Λ. These examples, Λ, are used to constrain the space of the candidate key-relations. SynthAttack, then, uses a verification check to confirm if the collected examples are sufficient to synthesize a valid key-relation. Otherwise, the counterexample from the failed verification check is identified as an `distinguishing input` (§2) to be added to Λ, and the algorithm repeats.

If there does not exist any distinguishing input for the locked circuit $\hat{\varphi}$, then $\hat{\varphi}$ has a unique semantic behavior—and that must be the correct functionality. Any key-relation that satisfies the counterexamples (distinguishing inputs) generated so far will be a valid candidate for the key relation. An inductive synthesis strategy based on distinguishing inputs allows us to quickly converge on a valid realization of the key-relation as each distinguishing input disqualifies many potential candidates for the key relation. Note that (as we illustrate the following example) there may be multiple, possibly semantically dissimilar, realizations of a key-relation that enables the same (correct) functionality on the locked circuit.

SynthAttack is outlined in Algorithm 2: the algorithm accepts the design of the locked circuit ($\hat{\varphi}$) and an activated circuit $\hat{\varphi}(\psi)$ (the locked circuit $\hat{\varphi}$ activated with a valid key-relation ψ). The notation $[\![\hat{\varphi}(\psi)]\!]$ indicates that this activated circuit can only be used as an input-output oracle, but cannot be inspected.

SynthAttack runs a counterexample-guided synthesis loop (Line 3). It checks for the existence of a distinguishing input in Line 4: if no such distinguishing input exists, it implies that the current set of examples is sufficient to synthesize a valid key-relation. So, in this case, the algorithm breaks out of the loop (Line 7-8) and synthesizes a key-relation (Line 16), that is returned as the synthesized, provably-correct instance for the key relation.

If there exists a distinguishing input X' (in Line 4), the algorithm uses the activated circuit to compute the expected output Y' corresponding to X (Line 10). This new counterexample (X', Y') is used to *block* all candidate key-relations that lead to an incorrect behavior, thereby reducing the potential choices for ψ_1 and ψ_2. The loop continues, again checking for the existence of distinguishing inputs on the updated constraint for Q_i.

The theoretical analysis of SynthAttack is available in the extended version [53]. The algorithm only terminates when it is able to synthesize a provably valid key-relation, that allows us to state the following result.

Theorem 2. *Algorithm 2 will always terminate, returning a key-relation ψ_1 such that $\hat{\varphi}(\psi_1)$ is semantically equivalent to $\hat{\varphi}(\psi)$, where ψ is the “correct" relation hidden by HOLL. (The proof is available in the extended version [53].)*

Example. SynthAttack on Fig. 1b iteratively generates six distinguishing inputs (Table 2). The key relation synthesized by SynthAttack (Fig. 6) is **not** semantically equivalent to the hidden key-relation that was computed and hidden by HOLL (Fig. 1c). This shows that there may exist multiple valid candidates for the key-relation that all evoke the same functionality on the locked design. For example, $X = \texttt{0100}$ generates $r_4 = \texttt{1}$ for the key relation in Fig. 1c but $r_4 = \texttt{0}$ for Fig. 6; however, the output of the locked circuit remains the same in both cases ($Y = \texttt{001}$).

$$\begin{aligned}&\{(r_0 \leftrightarrow x_1),\\ &(r_1 \leftrightarrow x_2),\\ &(r_2 \leftrightarrow r_0),\\ &(r_3 \leftrightarrow r_2 \wedge r_1),\\ &(r_4 \leftrightarrow \neg r_2 \wedge r_1)\}\end{aligned}$$

Fig. 6: Key relation generated by SynthAttack.

Table 2: Distinguishing inputs.

X	Y
1101	100
0001	001
0101	010
0111	100
1001	011
0011	011

6 Experimental Evaluation

We selected 100 combinational benchmarks from ISCAS’85 [1] and MCNC [58] and report the time for program synthesis and the overhead after applying our locking method. For long running experiments, we select a subset of 10 randomly selected benchmarks where, number of input ports range between 16 and 256, output ports range range between 7 and 245, `AND` gates in range [135, 4174].

For our experiments, we use number of relation terms as *budget* in the range `[12-14]` for the key relation and depth of expression selection in range `[2-4]`. We conduct our experiments on a machine with 32-Core Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz with 32GB RAM.

For both HOLL and SynthAttack, we use the Sketch synthesis tool. Since synthesis solvers are difficult to compare across different problem instances, we

were wary of the case where the defender gets an edge over the attacker due to use of different tools. We create the attack-team-defence-team asymmetry by controlling the computation time: while the defender gets 20 minutes (1200s) to generate locked circuit, the attacker runs the attack for up to 4 days.
Our experiments aim to answer five research questions:

RQ1. What is the attack resilience of HOLL? (§6.1)
RQ2. How do impact expression selection heuristics affect attack resilience? (§6.2)
RQ3. What is the hardware cost for HOLL? (§6.3)
RQ4. What is the time taken to synthesize the locked design and key-relation for HOLL? (refer to the extended version [53])
RQ5. What are the impact of the optimizations for scalability (backslicing and incremental synthesis)? (refer to the extended version [53])

Here is a summary of our findings:

Security. The key relations can be recovered completely by the attacker via SynthAttack but only for small circuits with a small hardware budget. For medium and large designs, key relations are fast to obtain (<1200s) but cannot be recovered by our attack even within 4 days. This shows our defense is efficient while our attack is strong but not scalable.

Hardware Cost. Our key relations with a budget of 12-14 latent terms have a minimal impact on the designs and the overhead reduces as the size of the circuit grows. On the largest benchmark, the area overhead is 1.2%. The corresponding configurations for programmable devices are small and provide high security.

HOLL Performance. The HOLL execution time ranges between 8s and 1001s, with an average of 33s for small, 17s for medium, and 60s for large designs for the budget of 8-10 latent terms. Our optimizations are crucial for the scalability of our HOLL defense (locking) algorithm: we fail to lock enough expressions in large circuits without these optimizations.

6.1 Attack Resilience

We define *attack resilience* of locked circuit, $\hat{\varphi}$, in terms of time taken to obtain a key relation, ψ', such that $\hat{\varphi} \wedge \psi'$ is equivalent to original circuits, φ.

Attack time. Fig. 8 shows the cumulative time spent till the i^{th} iteration (y-axis) of the loop versus the loop counter i, that is also the number of distinguishing inputs (samples) generated so far (x-axis). We show exponential *trend* curves (as a solid pink line) to capture the trend in the plotted points while the data-points are plotted as blue dots. The plots show that the plotted points follow the exponential trend lines, illustrating that SynthAttack does not scale well, thereby asserting the resilience of HOLL.

SynthAttack failed to construct a valid key-relation for any of these ten designs within a timeout of 4 days. However, for small designs with lesser number of latent terms, SynthAttack was able to construct a valid key-relation (Fig. 10).

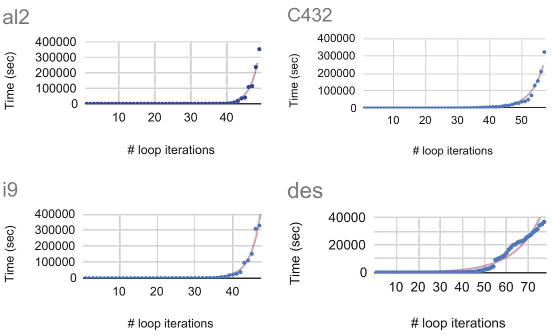


Fig. 8: Cumulative time for successive iterations of SynthAttack (best viewed in color)

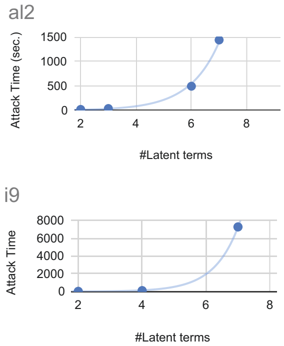


Fig. 10: Attack time vs #latent terms for `i9` and `al2`

Attack resilience vs. number of latent terms. The complexity of the key relation increases with the number of relation bits. As shown in Fig. 10 (for benchmarks `al2` and `i9`), the time required to break the locked circuit increases exponentially as the number of relation bits increases. We gave a timeout of `10` hours for this experiment and `al2` timed out at `9` latent terms, and `i9` timed out at `8` latent terms. Both results are for locked circuits with variables selected with the depth of locked expression, $\hat{e_i}$, equal to `1`.

6.2 Impact of Expression Selection on Attack Resilience

Attack resilience vs. Depth of locked expression. The attack resiliency of $\hat{\varphi}$ increases significantly as we increase the depth of the locked expression selected for HOLL for φ. We observe that for a number of latent terms in key relation equal to 2, for benchmark `al2`, increases from 213s to 3788s for depth 1 and 2, respectively. For benchmark `i9`, attack time increases from 351s to 1141s for depth 1 and 2, respectively.

Attack time vs. Coverage. To show the effect of coverage we select expressions (in $e_i \in E$) such that the distance (§3.2) among the expressions is largest (termed as diverse) and smallest (termed as converged). The attack time to break the locked circuit is more for diverse than converged expression selection heuristic. For example, for benchmarks `C432` and `i9`, attack time increases from 115s to 142s and 229s to 316s, respectively, when expression selection heuristic is changed from converged to diverse. The results are with three latent terms.

6.3 Hardware cost

The key relations can be implemented either as embedded Field Programmable Gate Array (eFPGA) or Programmable Array Logic. We synthesize the original and locked designs with Synopsys Design Compiler R-2020.09-SP1 targeting the Nangate `15`nm ASIC technology at standard operating conditions (`25`°C).

Table 3 provides the estimated cost for implementing the key relations with programmable devices. To do so, we compute the number of equivalent NAND2 gates used to estimate the number of 6-input LUTs. Given the number of LUTs, we give an estimation of the equivalent number of configuration bits (see [53] for details)–including those for switch elements. Results show that the size of the key relations is independent of original design size. Table 3 reports the fraction of the area locked with HOLL (*key relation*) to the area of the *original* circuit. The results show that the impact of HOLL is low, mainly for large designs.

Table 3: Hardware Impact of HOLL.

	Orig.	**Key Relation**			**Over-head**
Bench	**Area** (μm^2)	**Area** (μm^2)	**#Eq. LUT**	**#Eq. conf. bits**	**Area** **(%)**
`al2`	17.89	4.473	138	8,832	25.0
`cht`	20.74	4.178	132	8,448	20.1
`C432`	20.05	4.866	150	9,600	24.3
`C880`	50.04	4.325	132	8,448	8.6
`i9`	77.07	4.129	126	8,064	5.4
`i7`	80.41	4.129	126	8,064	4.3
`x3`	95.21	5.014	156	9,984	5.3
`frg2`	100.81	4.669	144	9,216	4.6
`i8`	120.37	4.325	132	8,448	3.6
`des`	445.37	5.554	174	11,136	1.2

7 Related Work

Logic Locking: Attacks and Defenses. Existing logic locking methods aptly operate on the gate-level netlists [54]. Gate-level locking cannot obfuscate all the semantic information because logic synthesis and optimizations absorb many of them into the netlist before the locking step. For example, constant propagation absorbs the constants into the netlist. Recently, alternative high-level locking methods obfuscate the semantic information before logic optimizations embed them into the netlist [37, 17]. For example, TAO applies obfuscations during HLS [37] but requires access to the HLS source code to integrate the obfuscations and cannot obfuscate existing IPs. Protecting a design at the register-transfer level (RTL) is an interesting compromise [29, 10]. Most of the semantic information (e.g., constants, operations, and control flows) is still present in the RTL and obfuscations can be applied to existing RTL IPs. In [29], the authors propose structural and functional obfuscation for DSP circuits. In [10], the authors propose a method to insert a special finite state machine to control the transition between obfuscated mode (incorrect function) and normal mode (correct function). Such transitions can only happen with a specific input sequence. Differently from [13], we extract the relation directly from the analysis of a single RTL design, making the approach independent of the design flow. None of these methods consider the possibility of hiding a relation among the key bits.

Program Synthesis. Program synthesis has been successful in many domains: synthesis of heap manipulations [39, 20, 57], bit-manipulating programs [27], bug synthesis [40], parser synthesis [30, 46], regression-free repairs [6, 5], synchronization in concurrent programs [56], boolean functions [22, 24, 23] and even differentially private mechanisms [38]. There has also been an interest in using program synthesis in hardware designs [16]. VeriSketch [4] exploits the power of program synthesis in hardware design. Our work is orthogonal to the objectives and techniques of VeriSketch: while VeriSketch secures hardware against timing attacks, we propose a hardware locking mechanism. Zhang et al. [62] use SyGUS based program synthesis to infer environmental invariants for verifying hardware circuits. We believe that this work shows the potential of applying programming languages techniques in hardware design. We believe that there is also a potential of applying program analysis techniques, symbolic [9, 21, 12, 36, 34], dynamic [41, 14] and statistical [28, 32, 11, 33], for hardware analysis; this is a direction we intend to pursue in the future.

8 Discussion

We end the paper with an important clarification: the *eFPGA configuration* in HOLL can also be represented as a bit sequence (i.e., a sequence of *configuration* bits). So, why can an attacker not launch attacks similar to SAT attacks on logic locking to recover the HOLL configuration bitstream?

The foremost reason is that while the key-bits in traditional logic locking simply represent a *value* that the attacker attempts to recover, the bit-sequence in HOLL is an encoding of a *program* [15, 35]. This raw bit-sequence used to program an eFPGA is too “low-level" to be synthesized directly—the size of such bit-streams is about **60-85** times of the keys used in traditional logic locking (128 key bit-sequence). So, the HOLL algorithm designer uses a higher-level domain-specific language (DSL) to synthesize the key relation (see §4), that is later “compiled" to the configuration sequence. The attacker will also have to use a similar strategy of using a high-level DSL to break HOLL.

However, while the designer of the key relation can use a well-designed small domain-specific language (DSL) that includes the exact set of components required (and a controlled budget) to synthesize the key relation, the attacker, not aware of the key relation or the DSL, will have to launch the attack with a “guess" of a large overapproximation. In other words, **the domain-specific language used for synthesis is also a *secret***, thereby making HOLL much harder to crack than traditional logic locking.

We evaluate HOLL (§6.1) under the assumption that the DSL (and budget) are known to the attacker. In real deployments (when the DSL is not known to the attacker), HOLL will be still more difficult to crack.

Acknowledgements. We thank the anonymous reviewers for their valuable inputs. The second author is supported in part by ONR grant number N000141812058. The last author wishes to thank Dr. Ramanuj Chouksey for his discussions on logic locking. We also wish to thank our colleagues who provided useful comments on this paper.

References

[1] ISCAS'85 benchmarks. https://filebox.ece.vt.edu/~mhsiao/iscas85.html, accessed: 2021-01-10

[2] Where is the eFPGA market and ecosystem headed? https://semiengineering.com/where-is-the-efpga-market-and-ecosystem-headed/, accessed: 2021-05-28

[3] Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Irlbeck, M., Peled, D.A., Pretschner, A. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015). https://doi.org/10.3233/978-1-61499-495-4-1

[4] Ardeshiricham, A., Takashima, Y., Gao, S., Kastner, R.: Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 1623–1638. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3354246

[5] Bavishi, R., Pandey, A., Roy, S.: Regression aware debugging for mobile applications. In: Mobile! 2016: Proceedings of the 1st International Workshop on Mobile Development (Invited Paper). p. 21–22. Mobile! 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/3001854.3001860

[6] Bavishi, R., Pandey, A., Roy, S.: To be precise: Regression aware debugging. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2984014

[7] Bhandari, J., Moosa, A.K.T., Tan, B., Pilato, C., Gore, G., Tang, X., Temple, S., Gaillardon, P.E., Karri, R.: Exploring eFPGA-based Redaction for IP Protection. In: International Conference on Computer-Aided Design (ICCAD) (Nov 2021)

[8] Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[9] Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. p. 209–224. OSDI'08, USENIX Association, USA (2008)

[10] Chakraborty, R.S., Bhunia, S.: RTL hardware IP protection using key-based control and data flow obfuscation. In: Proceedings of the International Conference on VLSI Design. pp. 405–410 (2010)

[11] Chatterjee, P., Chatterjee, A., Campos, J., Abreu, R., Roy, S.: Diagnosing software faults using multiverse analysis. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20. pp. 1629–1635. International Joint Conferences on Artificial Intelligence Organization (7 2020). https://doi.org/10.24963/ijcai.2020/226, main track

[12] Chatterjee, P., Roy, S., Diep, B.P., Lal, A.: Distributed bounded model checking. In: FMCAD (July 2020)

[13] Chen, J., Zaman, M., Makris, Y., Blanton, R.D.S., Mitra, S., Schafer, B.C.: DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual Property. In: Design Automation Conference (DAC). pp. 1–6 (2020)

[14] Chouhan, R., Roy, S., Baswana, S.: Pertinent path profiling: Tracking interactions among relevant statements. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–12 (2013). https://doi.org/10.1109/CGO.2013.6494983

[15] Clift, J., Murfet, D.: Encodings of turing machines in linear logic. Mathematical Structures in Computer Science **30**(4), 379–415 (2020)

[16] Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. In: 2009 Formal Methods in Computer-Aided Design. pp. 205–212 (2009). https://doi.org/10.1109/FMCAD.2009.5351120

[17] Di Crescenzo, G., Sengupta, A., Sinanoglu, O., Yasin, M.: Logic locking of boolean circuits: Provable hardware-based obfuscation from a tamper-proof memory. In: Simion, E., Géraud-Stewart, R. (eds.) Innovative Security Solutions for Information Technology and Communications. pp. 172–192. Springer International Publishing, Cham (2020)

[18] El Massad, M., Garg, S., Tripunitara, M.: Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In: Network and Distributed System Security Symposium (NDSS) (01 2015). https://doi.org/10.14722/ndss.2015.23218

[19] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (Jul 1987). https://doi.org/10.1145/24039.24041

[20] Garg, A., Roy, S.: Synthesizing heap manipulations via integer linear programming. In: Blazy, S., Jensen, T. (eds.) Static Analysis, SAS 2015. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_7

[21] Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065036

[22] Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for boolean function synthesis. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification (CAV). pp. 611–633. Springer International Publishing, Cham (2020)

[23] Golia, P., Roy, S., Meel, K.S.: Program synthesis as dependency quantified formula modulo theory. In: Zhou, Z.H. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21. pp. 1894–1900. International Joint Conferences on Artificial Intelligence Organization (8 2021). https://doi.org/10.24963/ijcai.2021/261, main Track

[24] Golia, P., Roy, S., Slivovsky, F., Meel, K.S.: Engineering an efficient boolean functional synthesis engine. In: ICCAD (2021)

[25] Guin, U., Huang, K., DiMase, D., Carulli, J.M., Tehranipoor, M., Makris, Y.: Counterfeit Integrated Circuits: A rising threat in the global semiconductor supply chain. Proceedings of the IEEE **102**(8), 1207–1228 (Aug 2014)

[26] Hurtarte, J., Wolsheimer, E., Tafoya, L.: Understanding Fabless IC Technology. Elsevier (Aug 2007)

[27] Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1806799.1806833

[28] Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of Test Information to Assist Fault Localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/581339.581397

[29] Lao, Y., Parhi, K.K.: Obfuscating DSP circuits via high-level transformations. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **23**(5), 819–830 (2015)

[30] Leung, A., Sarracino, J., Lerner, S.: Interactive parser synthesis by example. SIGPLAN Not. **50**(6), 565–574 (Jun 2015). https://doi.org/10.1145/2813885.2738002

[31] Li, L., Orailoglu, A.: Piercing logic locking keys through redundancy identification. In: Design, Automation and Test in Europe Conference (DATE). pp. 540–545 (2019)

[32] Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable Statistical Bug Isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065014

[33] Modi, V., Roy, S., Aggarwal, S.K.: Exploring Program Phases for Statistical Bug Localization. In: Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. PASTE '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2462029.2462034

[34] Pandey, A., Kotcharlakota, P.R.G., Roy, S.: Deferred concretization in symbolic execution via fuzzing. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 228–238. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330554

[35] Petersen, H.: Some remarks on real-time turing machines (2019), http://arxiv.org/abs/1902.00975

[36] Pham, V.T., Khurana, S., Roy, S., Roychoudhury, A.: Bucketing failing tests via symbolic analysis. In: Huisman, M., Rubin, J. (eds.) Fundamental Approaches to Software Engineering. pp. 43–59. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)

[37] Pilato, C., Regazzoni, F., Karri, R., Garg, S.: TAO: Techniques for algorithm-level obfuscation during high-level synthesis. In: Design Automation Conference (DAC). pp. 1–6 (Jun 2018)

[38] Roy, S., Hsu, J., Albarghouthi, A.: Learning differentially private mechanisms. In: 2021 2021 IEEE Symposium on Security and Privacy (SP). pp. 852–865. IEEE Computer Society, Los Alamitos, CA, USA (May 2021). https://doi.org/10.1109/SP40001.2021.00060

[39] Roy, S.: From concrete examples to heap manipulating programs. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_9

[40] Roy, S., Pandey, A., Dolan-Gavitt, B., Hu, Y.: Bug synthesis: Challenging bug-finding tools with deep faults. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 224–234. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236084

[41] Roy, S., Srikant, Y.N.: Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. p. 70–80. CGO ’09, IEEE Computer Society, USA (2009). https://doi.org/10.1109/CGO.2009.11

[42] S. W. Jones: Technology and Cost Trends at Advanced Nodes. IC Knowledge LLC (2019)

[43] Shamsi, K., Li, M., Meade, T., Zhao, Z., Pan, D.Z., Jin, Y.: Circuit obfuscation and oracle-guided attacks: Who can prevail? In: Great Lakes Symposium on VLSI. pp. 357–362. ACM, New York, NY, USA (2017)

[44] Shamsi, K., Li, M., Plaks, K., Fazzari, S., Pan, D.Z., Jin, Y.: IP protection and supply chain security through logic obfuscation: A systematic overview. ACM Transactions on Design Automation of Electronic Systems **24**(6) (Sep 2019)

[45] Shamsi, K., Pan, D.Z., Jin, Y.: On the impossibility of approximation-resilient circuit locking. In: IEEE International Symposium on Hardware Oriented Security and Trust. pp. 161–170 (2019)

[46] Singal, D., Agarwal, P., Jhunjhunwala, S., Roy, S.: Parse condition: Symbolic encoding of ll(1) parsing. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 57, pp. 637–655. EasyChair (2018). https://doi.org/10.29007/2ndp

[47] Sirone, D., Subramanyan, P.: Functional analysis attacks on logic locking. In: Design, Automation & Test Conference in Europe (DATE). pp. 1–6 (Mar 2019)

[48] Sisejkovic, D., Merchant, F., Reimann, L.M., Srivastava, H., Hallawa, A., Leupers, R.: Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach (2020)

[49] Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) Proceedings of Asian Symposium Programming Languages and Systems, 7th, (APLAS). vol. 5904, pp. 4–13. Springer (2009)

[50] Solar-Lezama, A.: Program sketching. vol. 15, p. 475–495. Springer-Verlag, Berlin, Heidelberg (Oct 2013). https://doi.org/10.1007/s10009-012-0249-7

[51] Subramanyan, P., Ray, S., Malik, S.: Evaluating the security of logic encryption algorithms. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 137–143 (2015)

[52] Takamaeda-Yamazaki, S.: Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL. In: arc. pp. 451–460 (Apr 2015)

[53] Takhar, G., Karri, R., Pilato, C., Roy, S.: HOLL: Program synthesis for higher order logic locking (2022), https://arxiv.org/abs/2201.10531

[54] Tan, B., Karri, R., Limaye, N., Sengupta, A., Sinanoglu, O., Rahman, M.M., Bhunia, S., Duvalsaint, D., Blanton, R., Rezaei, A., Shen, Y., Zhou, H., Li, L., Orailoglu, A., Han, Z., Benedetti, A., Brignone, L., Yasin, M., Rajendran, J., Zuzak, M., Srivastava, A., Guin, U., Karfa, C., Basu, K., Menon, V.V., French, M., Song, P., Stellari, F., Nam, G.J., Gadfort, P., Althoff, A., Tostenrude, J., Fazzari, S., Breckenfeld, E., Plaks, K.: Benchmarking at the frontier of hardware security: Lessons from logic locking (2020), https://arxiv.org/abs/2006.06806

[55] Venkatesh, G.A.: The semantic approach to program slicing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. p. 107–119. PLDI '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/113445.113455

[56] Verma, A., Kalita, P.K., Pandey, A., Roy, S.: Interactive debugging of concurrent programs under relaxed memory models. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. p. 68–80. CGO 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368826.3377910

[57] Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3106237.3106263

[58] Yang, S.: Logic synthesis and optimization benchmarks user guide: Version 3.0. Tech. rep., MCNC Technical Report (Jan 1991)

[59] Yasin, M., Mazumdar, B., Sinanoglu, O., Rajendran, J.: Removal attacks on logic locking and camouflaging techniques. IEEE Transactions on Emerging Topics in Computing **8**(2), 517–532 (2020)

[60] Yasin, M., Sengupta, A., Nabeel, M.T., Ashraf, M., Rajendran, J.J., Sinanoglu, O.: Provably-secure logic locking: From theory to practice. In: Conference on Computer and Communications Security. pp. 1601–1618 (2017)

[61] Yasin, M., Sengupta, A., Schafer, B.C., Makris, Y., Sinanoglu, O., Rajendran, J.: What to lock? functional and parametric locking. In: Proceedings of the on Great Lakes Symposium on VLSI 2017. pp. 351–356 (2017)

[62] Zhang, H., Yang, W., Fedyukovich, G., Gupta, A., Malik, S.: Synthesizing environment invariants for modular hardware verification. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **11990 LNCS**, 202–225 (2020). https://doi.org/10.1007/978-3-030-39322-9_10

The Complexity of LTL Rational Synthesis

Orna Kupferman and Noam Shenwald(✉)

School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel
orna@cs.huji.ac.il
noam.shenwald@mail.huji.ac.il

Abstract. In *rational synthesis*, we automatically construct a reactive system that satisfies its specification in all rational environments, namely environments that have objectives and act to fulfill them. We complete the study of the complexity of LTL rational synthesis. Our contribution is threefold. First, we tighten the known upper bounds for settings that were left open in earlier work. Second, our complexity analysis is parametric, and we describe tight upper and lower bounds in each of the problem parameters: the game graph, the objectives of the system components, and the objectives of the environment components. Third, we generalize the definition of rational synthesis, combining the cooperative and non-cooperative approaches studied in earlier work, and extend our complexity analysis to the general definition.

1 Introduction

Synthesis is the automated construction of a system from its specification. The basic idea is simple and appealing: instead of developing a system and verifying that it adheres to its specification, we use an automated procedure that, given a specification, constructs a system that is correct by construction, thus enabling the designers to focus on *what* the system should do rather than *how* to do it. A reactive system interacts with its environment and should satisfy its specification in all environments [8,25]. Accordingly, synthesis corresponds to a *zero-sum game* between the system and the environment, where they together generate a computation, the system wins if the computation satisfies the specification, and otherwise, the environment wins.

In practice, the requirement to satisfy the specification in all environments is often too strong. Therefore, it is common to add assumptions on the environment. An assumption may be direct, say a specification that restricts the possible behaviors of the environment [5], or less direct, say a bound on the size of the environment or other resources it uses [14]. In [11], the authors suggest a conceptual assumption on the environment, namely its rationality: *Rational synthesis* is based on the idea that the components composing the environment typically have objectives of their own, and they act to achieve their objectives. For example, clients interacting with a server typically have objectives other

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 25–45, 2022.
https://doi.org/10.1007/978-3-030-99524-9_2

than to fail the server. As shown in [11], the system can capitalize on the rationality and objectives of components that compose its environment. Adding rationality into the picture makes the corresponding game *non-zero-sum* [22], thus objectives of different players may overlap.

The interesting questions about non-zero-sum games concern *stable outcomes*, in particular *Nash equilibria* (NE) [21]. More formally, each of the players in the game has a *strategy* that directs her which actions to take; a *profile* is a vector of strategies, one for each player; each profile has an *outcome* (in our case, the computation generated when the system and the environment follow their strategies); and a profile is an NE if no player has an incentive to deviate from it (in our case, to change her strategy in a way that would cause the outcome of the new profile to satisfy her objective).

Two approaches to rational synthesis have been studied. In *cooperative* rational synthesis (CRS) [11], the desired output is an NE profile whose outcome satisfies the objective of the system. Thus, in CRS, we assume that we can suggest strategies to the environment players, and once they have no incentive to deviate from these strategies, they follow them. Then, in *non-cooperative* rational synthesis (NRS) [15], the desired output is a strategy for the system player such that the objective of the system is satisfied in the outcome of all NE profiles that include this strategy. Thus, in NRS, the environment players are rational, but we cannot suggest them a strategy.

The cooperative and non-cooperative approaches correspond to different settings in reality, having to do both with the technical ability to communicate a strategy to the environment players, say due to different architectures, as well as the willingness of the environment players to follow a suggested strategy. As shown in [1], the two approaches are related to the two stability-inefficiency measures of *price of stability* [3] and *price of anarchy* [16,23]. Additional related work includes *rational verification* [27,12], where we check that a given system satisfies its specification when interacting with a rational environment, and extensions of rational synthesis to richer settings (multi-valued, partial visibility, and more) [4,13,18].

The *complexity* of rational synthesis was first studied for the case the input to the problem is the objectives of the players, given by LTL formulas. In this setting, CRS is in 2EXPTIME [11], whereas the best known upper bound for NRS until recently was 3EXPTIME [15] (the paper specifies a 2EXPTIME upper bound, but a careful analysis of the algorithm reveals that it is actually in 3EXPTIME), improved to 2EXPTIME for *turn-based* games with two players [18]. The complexity analysis above suggests that rational synthesis is not harder than traditional synthesis. One may wonder whether this has to do with the doubly-exponential translation of LTL to deterministic automata, which dominates the complexity. To answer this question, [9] studies the complexity of rational synthesis where the objectives of the players are given by ω-regular winning conditions in a game graph (e.g., reachability, Büchi, and parity). The analysis in [9] also distinguishes between the case the number of players is fixed and the case it is not. As shown there, in most cases the complexity of the rational variant coincides with the complexity of the zero-sum game. In some cases, however, it does not. For example, while the problem of deciding Rabin games is NP-complete [10], the best algorithm for solving CRS with Rabin objectives is in P^{NP}, going up to PSPACE-complete in NRS, and going higher when the number of players is not fixed [9].

In this work, we complete the study of the complexity of LTL rational synthesis. Our contribution is threefold. First, we tighten the known upper bound for NRS for settings with three or more players and for *concurrent* games, which were left open in [9,18]. Second, our complexity analysis is *parametric*, and we describe tight upper and lower bounds in each of the problem parameters: the game graph, the objectives of the system players, and the objectives of the environment players. Third, we generalize the definition of rational synthesis, combining the cooperative and non-cooperative approaches, and extend our complexity analysis to the general definition. Below we elaborate on each of the contributions.

Let us start with the generalization of the problem. In our general definition, we may suggest a strategy only to a subset of the environment players. Thus, we distinguish between three types of players: controllable, cooperative uncontrollable, and non-cooperative uncontrollable. Then, in the (general) rational-synthesis (RS) problem, we are given a labeled graph and LTL formulas that specify the objectives of the players, and we seek strategies for the controllable and the cooperative-uncontrollable players such that the objectives of the controllable players are satisfied in the outcome of every NE profile that extends these strategies. Note that CRS and NRS can be viewed as special cases of RS where the uncontrollable players are all cooperative or all non-cooperative.

In the tight-complexity front, our algorithms reduce rational synthesis to the nonemptiness problem of tree automata. The automata accept *certified strategy trees*: trees that are labeled by both a strategy for the controllable player[1] and information about uncontrollable players that deviate and the strategies to which they deviate. The most technically-challenging algorithm we describe is for NRS in the concurrent setting. While in the turn-based setting, we need a single player that deviates in order to justify a path in which the objective of the controllable player is not satisfied, in the concurrent setting, where the players choose actions simultaneously and independently, we need to consider sets of uncontrollable players. This makes the certificate much more complex. In particular, it involves labels from an exponential alphabet, which introduces an additional challenge, namely a need to decompose labels along branches in the tree. Also, while in the turn-based setting, an NE always exists, concurrent games with three or more players need not have an NE [7], and so a certified strategy tree should also certify the existence of an NE.

Finally, in the parameterized-complexity front, the fact our algorithms use tree automata (rather than a translation to Strategy Logic [6], which has been the case in [11,15]), enables us to analyze the complexity in each of the parameters of the problem: the game graph G, the objective ψ_1 of the controllable player, and the objectives $\psi_2, \ldots, \psi_k$ of the uncontrollable players. For CRS, [18] studies the

[1] It is easy to see that several controllable components can be merged to a single one.

parameterized complexity in turn-based games with two players.[2] The algorithm there is based on a distinction between the case the uncontrollable player satisfies her objective and the case she does not. Generalizing this to an arbitrary number of players, we parameterize solutions with the set of the uncontrollable players whose objectives are satisfied, and give a uniform solution to all cases. This also enables us to seek solutions that favor some or all uncontrollable players.

We show that the complexity of CRS is polynomial in $|G|$, doubly-exponential in $|\psi_2|, \ldots, |\psi_k|$, and only exponential in $|\psi_1|$. Thus, in terms of the system specification, CRS is in fact easier than traditional synthesis! Once we move to NRS or RS, the complexity becomes doubly exponential in all objectives. We describe tight lower bounds for the different parameters, and we show that they are valid already for the case $k = 2$ and the game is turn based. Specifically, we prove that CRS is EXPTIME-hard even when G and ψ_2 are fixed, and is 2EXPTIME-hard even when G and ψ_1 are fixed. Similarly, NRS is 2EXPTIME-hard even when only one of ψ_1 and ψ_2 is not fixed. In order to see the technical challenge in our lower-bound proofs, consider the current 2EXPTIME lower-bound proof for CRS, where synthesis of an objective ψ for the system is reduced to CRS with objectives ψ for the system and $\neg\psi$ for the environment. The reduction crucially depends on both objectives not being fixed, and just changing either of them to *True* or *False* does not do the trick. In order to get 2EXPTIME-hardness in $|\psi_2|$, we need to cleverly manipulate both G and ψ_1.

Together, our results complete the complexity picture for a generalized definition of rational synthesis, for both turn-based and concurrent systems, with any number of components, and with the exact dependencies in each of the parameters of the problem.

Due to the lack of space, some proofs are omitted and can be found in the full version, in the authors' URLs.

2 Preliminaries

2.1 LTL, trees, and automata

The logic LTL is used for specifying on-going behaviors of reactive systems [24]. Formulas of LTL are constructed from a set AP of atomic propositions using the usual Boolean operators and the temporal operators G ("always") and F ("eventually"), X ("next time") and U ("until"). The semantics of LTL is defined with respect to infinite computations in $(2^{AP})^\omega$. We are going to use LTL for specifying the objectives of the system and the components composing the environment.

Given a set D of directions, a *D-tree* is a set $T \subseteq D^*$ such that if $x \cdot d \in T$, where $x \in D^*$ and $d \in D$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $x \in T$, the nodes $x \cdot d$, for

[2] The study in [18] considers *perspective games* [19], which adds the challenge of *partial visibility* on top of rational synthesis, but the results there imply the desired bounds for the case of full visibility.

$d \in D$, are the *successors* of x, and the *direction* of node $x \cdot d$ is d. A *path* h in a tree T is a set $h \subseteq T$ such that $\varepsilon \in h$ and for every $x \in h$, either x is a leaf or there exists a unique $d \in D$ such that $x \cdot d \in h$. We sometimes refer to paths in T as words in D^* or D^ω. For a finite path $h \subseteq D^*$ and a finite or infinite path $h' \subseteq D^*$, we use $h \preceq h'$ to indicate that h is a prefix of h', thus $h \subseteq h'$. Given an alphabet Σ, a *Σ-labeled D-tree* is a pair $\langle T, \tau \rangle$ where T is a tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ.

Our algorithms use *automata on infinite words and trees.* We are going to use nondeterministic and universal automata, yet define below alternating automata, which subsume both classes. For a set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**. For a set $Y \subseteq X$ and a formula $\theta \in \mathcal{B}^+(X)$, we say that Y *satisfies* θ iff assigning **true** to elements in Y and assigning **false** to elements in $X \setminus Y$ makes θ true. An *alternating tree automaton* is $\mathcal{A} = \langle \Sigma, D, Q, q_{in}, \delta, \alpha \rangle$, where Σ is the input alphabet, D is a set of directions, Q is a finite set of states, $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$ is a transition function, $q_{in} \in Q$ is an initial state, and $\alpha \subseteq Q$ specifies a Büchi or a co-Büchi acceptance condition. For a state $q \in Q$, we use $\mathcal{A}^q$ to denote the automaton obtained from $\mathcal{A}$ by setting the initial state to be q. The *size* of $\mathcal{A}$, denoted $|\mathcal{A}|$, is the sum of lengths of formulas that appear in δ.

The alternating automaton $\mathcal{A}$ runs on Σ-labeled D-trees. A *run* of $\mathcal{A}$ over a Σ-labeled D-tree $\langle T, \tau \rangle$ is a $(T \times Q)$-labeled $\mathbb{N}$-tree $\langle T_r, r \rangle$. Each node of T_r corresponds to a node of T. A node in T_r, labeled by (x, q), describes a copy of the automaton that reads the node x of T and visits the state q. Note that many nodes of T_r can correspond to the same node of T. The labels of a node and its successors have to satisfy the transition function. Formally, $\langle T_r, r \rangle$ satisfies the following:

1. (1) $\varepsilon \in T_r$ and $r(\varepsilon) = \langle \varepsilon, q_{in} \rangle$.
2. (2) Let $y \in T_r$ with $r(y) = \langle x, q \rangle$ and $\delta(q, \tau(x)) = \theta$. Then there is a (possibly empty) set $S = \{(c_0, q_0), (c_1, q_1), \ldots, (c_{n-1}, q_{n-1})\} \subseteq D \times Q$, such that S satisfies θ, and for all $0 \leq i \leq n-1$, we have $y \cdot i \in T_r$ and $r(y \cdot i) = \langle x \cdot c_i, q_i \rangle$.

For example, if $\langle T, \tau \rangle$ is a $\{0,1\}$-tree with $\tau(\varepsilon) = a$ and $\delta(q_{in}, a) = ((0, q_1) \vee (0, q_2)) \wedge ((0, q_3) \vee (1, q_2))$, then, at level 1, the run $\langle T_r, r \rangle$ includes a node labeled $(0, q_1)$ or a node labeled $(0, q_2)$, and includes a node labeled $(0, q_3)$ or a node labeled $(1, q_2)$. Note that if, for some y, the transition function δ has the value **true**, then y need not have successors. Also, δ can never have the value **false** in a run.

A run $\langle T_r, r \rangle$ is accepting if all its infinite paths satisfy the acceptance condition. Given a run $\langle T_r, r \rangle$ and an infinite path $\pi \subseteq T_r$, let $inf(\pi) \subseteq Q$ be such that $q \in inf(\pi)$ if and only if there are infinitely many $y \in \pi$ for which $r(y) \in T \times \{q\}$. That is, $inf(\pi)$ contains exactly all the states that appear infinitely often in π. A path π satisfies a *Büchi* acceptance condition α iff $inf(\pi) \cap \alpha \neq \emptyset$, and satisfies a *co-Büchi* acceptance condition α iff $inf(\pi) \cap \alpha = \emptyset$. We also consider the *parity* acceptance condition, where $\alpha : Q \to \{0, 1, \ldots, k\}$ maps each state to a color in $\{0, 1, \ldots, k\}$, and a path π satisfies α if the minimal color visited infinitely often is even, thus $\min\{i : \mathit{inf}(\pi) \cap \alpha^{-1}(i) \neq \emptyset\}$ is even. An automaton accepts a tree iff there exists a run that accepts it. We denote by $L(\mathcal{A})$ the set of all Σ-labeled trees that $\mathcal{A}$ accepts. The *size* of $\mathcal{A}$, denoted $|\mathcal{A}|$, is the sum of lengths of the description of its transition function.

The alternating automaton $\mathcal{A}$ is *nondeterministic* if for all the formulas that appear in δ, if (c_1, q_1) and (c_2, q_2) are conjunctively related, then $c_1 \neq c_2$. (i.e., if the transition is rewritten in disjunctive normal form, there is at most one element of $\{c\} \times Q$, for each $c \in D$, in each disjunct). Note that then, the run tree T_r is equal to T, and the Σ-labels in τ are replaced by Q-labels in r. The automaton $\mathcal{A}$ is *universal* if all the formulas that appear in δ are conjunctions of atoms in $D \times Q$. Note that then, there is only one run tree of $\mathcal{A}$ on $\langle T, \tau \rangle$, yet each note $x \in T$ may have several nodes $y \in T_r$ such that $r(y) = \langle x, q \rangle$ for some $q \in Q$. Finally, $\mathcal{A}$ is *deterministic* if it is both nondeterministic and universal. The automaton $\mathcal{A}$ is a *word* automaton if $|D| = 1$. Then, we can omit D from the specification of the automaton and denote the transition function of $\mathcal{A}$ as $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$. If the word automaton is nondeterministic or universal, then $\delta : Q \times \Sigma \to 2^Q$.

We denote different types of automata by three-letter acronyms in $\{D, N, U\} \times \{F, B, C, P\} \times \{W, T\}$, where the first letter describes the branching mode of the automaton (deterministic, nondeterministic, or universal), the second letter describes the acceptance condition (finite, Büchi, co-Büchi, or parity), and the third letter describes the object over which the automaton runs (words or trees). For example, UCT stands for a universal co-Büchi tree automaton.

2.2 Concurrent multiplayer games

For $k \geq 1$, let $[k] = \{1, \ldots, k\}$. A *k-player game graph* is a tuple $G = \langle AP, V, v_0, \{A_i\}_{i \in [k]}, \{\kappa_i\}_{i \in [k]}, \delta, \tau \rangle$, where AP is a set of atomic propositions, V is a set of vertices, $v_0 \in V$ is an initial vertex, and for $i \in [k]$, the set A_i is a set of actions of Player i, and $\kappa_i : V \to 2^{A_i}$ specifies the set of actions that Player i can take at each vertex.

A *move* in G is a tuple $\langle a_1, \ldots, a_k \rangle \in A_1 \times \cdots \times A_k$, describing possible choices of actions for all k players. A move $\langle a_1, \ldots, a_k \rangle$ is *possible* for vertex $v \in V$ if $a_i \in \kappa_i(v)$ for all $i \in [k]$. Then, the transition function $\delta : V \times A_1 \times \cdots \times A_k \to V$ is a deterministic function that maps each vertex and possible move for it to a successor vertex. Finally, the function $\tau : V \to 2^{AP}$ maps each vertex to the set of atomic propositions that hold in it.

A *game* is a tuple $\mathcal{G} = \langle G, \{\psi_i\}_{i \in [k]} \rangle$, where G is a k-player game graph, and ψ_i, for $i \in [k]$, is an LTL formula over AP, describing the *objective* of Player i. In a beginning of a play in the game, a token is placed on v_0. Then, at each round, the players choose actions simultaneously and independently of the other players, and the induced move determines the successor vertex. Repeating this, the players generate a *play* $\rho = v_0, v_1, \ldots$ in G, which induces the *computation* $\tau(\rho) = \tau(v_0), \tau(v_1), \ldots \in (2^{AP})^\omega$. For every $i \in [k]$, Player i aims for a play whose computation satisfies ψ_i. For an LTL formula ψ, let $L(\psi) \subseteq (2^{AP})^\omega$ be the set of computations that satisfy ψ.

A *strategy* for Player i is a function $f_i : V^+ \to A_i$ that maps histories of the game to an action suggested to Player i. The suggestion has to be consistent with κ_i. Thus, for every $v_0 v_1 \cdots v_j \in V^+$, we have that $f_i(v_0 v_1 \cdots v_j) \in \kappa_i(v_j)$. A *profile* is a tuple $\pi = \langle f_1, \ldots, f_k \rangle$ of strategies, one for each player. The *outcome* of a profile $\pi = \langle f_1, \ldots, f_k \rangle$ is the play obtained when the players follow their strategies. Formally, $\mathsf{Outcome}(\pi) = v_0, v_1, \ldots$ is such that for all $j \geq 0$, we have that $v_{j+1} = \delta(v_j, \langle f_1(v_0 \cdots v_j), \cdots, f_k(v_0 \cdots v_j) \rangle)$. For a subset $S \subseteq [k]$ of players, an *S-profile* is a set of strategies, one for each player in S. We say that a profile π *extends* an S-profile π' if the players in S use in π their strategies in π'.

Consider a profile π. The set of *winners* in π, denoted $\mathsf{Win}(\pi)$, is the set of players whose objectives are satisfied in $\mathsf{Outcome}(\pi)$. Formally, $i \in \mathsf{Win}(\pi)$ iff $\tau(\mathsf{Outcome}(\pi)) \in L(\psi_i)$. The set of *losers* in π, denoted $\mathsf{Lose}(\pi)$, is then $[k] \setminus \mathsf{Win}(\pi)$, namely the set of players whose objectives are not satisfied in $\mathsf{Outcome}(\pi)$.

A game $\mathcal{G}$ is *zero-sum* if the objectives of the players form a partition of all possible behaviors. That is, for every $i \neq j \in [k]$, we have that $L(\psi_i) \cap L(\psi_j) = \emptyset$, and $\bigcup_{i \in [k]} L(\psi_i) = (2^{AP})^\omega$. Accordingly, for every profile π in a zero-sum game, we have that $|\mathsf{Win}(\pi)| = 1$ and $|\mathsf{Lose}(\pi)| = k-1$. We then say that Player i *wins* $\mathcal{G}$ if she has a *winning strategy* – a strategy that guarantees the satisfaction of ψ_i no matter how the other players proceed. Formally, f_i is a winning strategy if for every profile π with f_i, we have that $\mathsf{Win}(\pi) = \{i\}$.

Games may be *non zero-sum*, thus the objectives of the players may overlap. In such games, we are interested in *stable* profiles. In particular, a profile $\pi = \langle f_1, \ldots, f_k \rangle$ is a *Nash Equilibrium* (NE, for short) [21] if, intuitively, no (single) player can benefit from unilaterally changing her strategy. In our setting, benefiting amounts to moving from the set of losers to the set of winners. Formally, for $i \in [k]$ and a strategy f'_i for Player i, let $\pi[i \leftarrow f'_i] = \langle f_1, \ldots, f_{i-1}, f'_i, f_{i+1}, \ldots, f_k \rangle$ be the profile obtained from π by changing the strategy of Player i to f'_i. We say that π is an NE if for every $i \in [k]$, if $i \in \mathsf{Lose}(\pi)$, then for every strategy f'_i, we have that $i \in \mathsf{Lose}(\pi[i \leftarrow f'_i])$. Thus, π is an NE if no player has an incentive to deviate from π. For a subset $\mathsf{W} \subseteq [k]$ of players, we say that π is a W-*NE* if π is an NE with $\mathsf{W} = \mathsf{Win}(\pi)$.

The game $\mathcal{G}$ is *turn-based* if the transition function of its graph G is such that for every vertex $v \in V$, there is a single player that "owns" v and determines the successor vertex whenever the play is in v. Formally, for every $v \in V$, there is $i \in [k]$ such that for all moves $\langle a_1, \ldots, a_k \rangle$ and $\langle a'_1, \ldots, a'_k \rangle$ that are possible for v, if $a_i = a'_i$, then $\delta(v, \langle a_1, \ldots, a_k \rangle) = \delta(v, \langle a'_1, \ldots, a'_k \rangle)$. Accordingly, we describe the game graph of a turn-based game as $G = \langle AP, \{V_i\}_{i \in [k]}, v_0, E, \tau \rangle$, where $V_1, \ldots, V_k$ is a partition of V to the sets of vertices owned by the different players, and $E \subseteq V \times V$ is the transition relation, modeling the fact that the set of actions of Player i in a vertex v she owned is the set of v's successors.

3 Rational Synthesis

Consider a k-player game $\mathcal{G} = \langle G, \{\psi_i\}_{i\in[k]}\rangle$. We distinguish between three types of players: A player is *controllable* if she is guaranteed to follow a strategy assigned to her. Otherwise, she is *uncontrollable.* The uncontrollable players are *rational* – they would not deviate from a profile unless they have a beneficial deviation from it. We distinguish between *cooperative uncontrollable* players, to which we can suggest a strategy (which they would follow unless they have a beneficial deviation), and *non-cooperative uncontrollable* players, to which we cannot suggest a strategy. The distinction between the cooperative and non-cooperative uncontrollable players may be induced by the architecture or the nature of the players. We denote by C, CU, and NU the disjoint partition of $[k]$ into the classes of controllable, uncontrollable cooperative, and uncontrollable non-cooperative players, respectively.

In rational synthesis, we seek a strategy for each of the players in C with which their objectives are guaranteed to be satisfied, assuming rationality of the other players. As we have the best interest of the players in C in mind, we assume that $\mathsf{C} \neq \emptyset$. We say that a profile $\pi = \langle f_1, \ldots, f_k\rangle$ is a C*-fixed NE*, if no player in $\mathsf{CU} \cup \mathsf{NU}$ has a beneficial deviation. Formally, we have the following.

Definition 1. [Rational Synthesis] *Consider a k-player game $\mathcal{G} = \langle G, \{\psi_i\}_{i\in[k]}\rangle$. The problem of rational synthesis (RS) is to return a* $(\mathsf{C}\cup\mathsf{CU})$*-profile π' such that there is a* C*-fixed NE that extends π', and for every* C*-fixed NE π that extends π', we have that* $\mathsf{C} \subseteq \mathsf{Win}(\pi)$*.*

Two special cases of rational synthesis have been studied in the literature. The first is *cooperative rational synthesis*, where all uncontrollable players are cooperative [11]. The second is *non-cooperative rational synthesis*, where all uncontrollable players are non-cooperative [15].

Definition 2. [Cooperative Rational Synthesis] *Consider a k-player game $\mathcal{G} = \langle G, \{\psi_i\}_{i\in[k]}\rangle$ with* $\mathsf{NU} = \emptyset$*. The problem of cooperative rational synthesis (CRS) is to return a* C*-fixed NE π such that* $\mathsf{C} \subseteq \mathsf{Win}(\pi)$*.*

Definition 3. [Non-Cooperative Rational Synthesis] *Consider a k-player game $\mathcal{G} = \langle G, \{\psi_i\}_{i\in[k]}\rangle$ with* $\mathsf{CU} = \emptyset$*. The problem of non-cooperative rational synthesis (NRS) is to return a* C*-profile π' such that there is a* C*-fixed NE that extends π', and for every* C*-fixed NE π that extends π', we have that* $\mathsf{C} \subseteq \mathsf{Win}(\pi)$*.*

Remark 1. The original rational synthesis problem does not include a game graph [11]. Instead, the set AP over which the objectives are defined is partitioned among the players, and at each round of the game, each player chooses an assignment to the subset of AP she controls. It is easy to see that this setting is a special case of our setting, taking the graph to have vertices in 2^{AP}. □

Remark 2. In previous work, the definition of NRS does not require the existence of a C-fixed NE that extends π' [15,18]. In some settings (in particular, turn-based games), the existence of such an NE is guaranteed. In others (in particular, concurrent games) there need not be an NE in games with three or more players [7]. Note, however, that even with the requirement that a C-fixed NE that extends π' exists, there is no guarantee that best response dynamics from π' would lead to such a C-fixed NE. □

As in traditional synthesis, one can also define the corresponding decision problems, of *rational realizability*, where we only need to decide whether the desired strategies exist. In order to avoid additional notations, we sometimes refer to RS, CRS, and NRS also as decision problems.

For a set $\mathsf{W} \subseteq [k]$, we say that a solution to the rational synthesis problem is a W-*solution* iff it is a solution that guarantees the winning of exactly the players in W. In particular, a $(\mathsf{C} \cup \mathsf{CU})$-profile π' is a W-*RS solution* if it is an RS solution such that for every C-fixed NE π that extends π', we have that $\mathsf{W} = \mathsf{Win}(\pi)$; a profile π is a W-*CRS solution* if π is a CRS solution such that $\mathsf{W} = \mathsf{Win}(\pi)$; and a C-profile π' is a W-*NRS solution* if π' is an NRS solution such that for every C-fixed NE π that extends π', we have that $\mathsf{W} = \mathsf{Win}(\pi)$.

It is easy to see that since the players in C are controllable, we can treat them as a single player with an objective that is the conjunction of the objectives of the players in C. Accordingly, in the sequel we assume that $\mathsf{C} = \{1\}$.

Remark 3. It is easy to add to the setting *uncontrollable hostile* players, namely players that, as in traditional synthesis, do not have an objective. Indeed, an uncontrollable hostile player is equivalent to an uncontrollable (cooperative or non-cooperative) player with objective $\neg\psi_1$. □

4 The Complexity of Cooperative Rational Synthesis

In this section we study the complexity of CRS. Consider a k-player concurrent game $\mathcal{G} = \langle G, \{\psi_i\}_{i \in [k]} \rangle$. A strategy for Player i can be viewed as an A_i-labeled V-tree, and a profile can be viewed as an $(A_1 \times \cdots \times A_k)$-labeled V-tree. Formally, if $\pi = \langle f_1, \ldots, f_k \rangle$ then for every node $h \in V^*$ in the *profile tree* $\langle V^*, \pi \rangle$, we have $\pi(h) = \langle f_1(h), \ldots, f_k(h) \rangle$, where $\langle V^*, f_i \rangle$ is the *strategy tree* that corresponds to f_i. Note that $\mathsf{Outcome}(\pi)$ then corresponds to a path in $\langle V^*, \pi \rangle$.

Viewing profiles as labeled trees enables us to reduce CRS to the nonemptiness of a tree automaton. Essentially, the automaton accepts all profile trees that are solutions to the CRS problem. We define the automaton by decomposing the solutions according to the set of players that win. Given a set W of players with $1 \in \mathsf{W}$, a profile π is a W-CRS solution iff π is a 1-fixed W-NE. Thus, iff exactly the players in W win in $\mathsf{Outcome}(\pi)$, and for every $i \notin \mathsf{W}$, Player i loses in $\mathsf{Outcome}(\pi[i \leftarrow f'_i])$, for every strategy f'_i. In Theorem 1 below, we construct automata that check these conditions.

Theorem 1. *Consider a set of players* W *with* $1 \in \mathsf{W}$. *We can construct the following tree automata over* $(A_1 \times \cdots \times A_k)$*-labeled* V*-trees:*

- *An NBT* $\mathcal{N}_{\mathsf{W}}$ *that accepts a profile tree* $\langle V^*, \pi \rangle$ *iff* $\mathsf{Win}(\pi) = \mathsf{W}$. *The size of* $\mathcal{N}_{\mathsf{W}}$ *is polynomial in* $|G|$ *and exponential in* $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$.

- *For every $i \notin \mathsf{W}$, a UCT $\mathcal{U}^i_{\mathsf{W}}$ that accepts a profile tree $\langle V^*, \pi\rangle$ iff $i \in \mathsf{Lose}(\pi[i \leftarrow f'_i])$, for every strategy f'_i. The size of $\mathcal{U}^i_{\mathsf{W}}$ is polynomial in $|G|$ and exponential in $|\psi_i|$.*

Proof. We start with the NBT $\mathcal{N}_{\mathsf{W}}$. Recall that we want $\mathcal{N}_{\mathsf{W}}$ to accept a profile tree $\langle V^*, \pi\rangle$ iff exactly the players in W win in $\mathsf{Outcome}(\pi)$. Let $\psi = \bigwedge_{i \in \mathsf{W}} \psi_i \wedge \bigwedge_{i \notin \mathsf{W}}(\neg\psi_i)$, and let $\mathcal{A} = \langle 2^{AP}, Q, q_0, \mu, \alpha\rangle$ be an NBW of size exponential in $|\psi|$ that corresponds to ψ. The NBT $\mathcal{N}_{\mathsf{W}}$ follows the outcome in the profile tree, and checks if the players in W are exactly the winners of the profile. Formally, $\mathcal{N}_{\mathsf{W}} = \langle A_1 \times \cdots \times A_k, V, V \times Q, \langle v_0, q_0\rangle, \eta, V \times \alpha\rangle$, where for every $\langle v, q\rangle \in V \times Q$ and $\langle a_1, \ldots, a_k\rangle \in A_1 \times \cdots \times A_k$, we have that $\eta(\langle v, q\rangle, \langle a_1, \ldots, a_k\rangle) = \bigvee_{q' \in \mu(q, \tau(v))}(\delta(v, \langle a_1, \ldots, a_k\rangle), \langle \delta(v, \langle a_1, \ldots, a_k\rangle), q'\rangle)$.

We continue to the UCT $\mathcal{U}^i_{\mathsf{W}}$. Recall that we want $\mathcal{U}^i_{\mathsf{W}}$ to accept a profile tree $\langle V^*, \pi\rangle$ iff Player i loses in $\mathsf{Outcome}(\pi[i \leftarrow f'_i])$, for every strategy f'_i. Let $\mathcal{U}_i = \langle 2^{AP}, Q_i, q^0_i, \delta_i, \alpha_i\rangle$ and $\neg\mathcal{U}_i = \langle 2^{AP}, S_i, s^0_i, \mu_i, \beta_i\rangle$ be the UCWs corresponding to ψ_i and $\neg\psi_i$, respectively. The UCT $\mathcal{U}^i_{\mathsf{W}}$ follows every possible deviation for Player i, and checks that indeed she always loses. Formally, $\mathcal{U}^i_{\mathsf{W}} = \langle A_1 \times \cdots \times A_k, V, V \times S_i, \langle v_0, s^0_i\rangle, \eta, V \times \beta_i\rangle$, where for every $\langle v, s\rangle \in V \times S_i$ and $\langle a_1, \ldots, a_k\rangle \in A_1 \times \cdots \times A_k$, we have that $\eta(\langle v, s\rangle, \langle a_1, \ldots, a_k\rangle) = \bigwedge_{a'_i \in \kappa_i(v)} \bigwedge_{s' \in \mu_i(s, \tau(v))}(\delta(v, \langle a_1, \ldots, a'_i, \ldots, a_k\rangle), \langle \delta(v, \langle a_1, \ldots, a'_i, \ldots, a_k\rangle), s'\rangle)$. □

Theorem 2. *Solving CRS can be done in time polynomial in $|G|$, exponential in $|\psi_1|$, and doubly-exponential in $|\psi_2|, \ldots, |\psi_k|$. The problem is EXPTIME-hard in $|\psi_1|$ and 2EXPTIME-hard in each of $|\psi_2|, \ldots, |\psi_k|$.*

Proof. We start with the upper bound. It is easy to see that for every set $\mathsf{W} \subseteq [k]$ of players with $1 \in \mathsf{W}$, there is a W-CRS solution iff the intersection of the automata constructed in Theorem 1 is nonempty. We construct an NBT $\mathcal{A}$ such that $L(\mathcal{A}) \neq \emptyset$ iff $L(\mathcal{N}_{\mathsf{W}}) \cap \bigcap_{i \notin \mathsf{W}} L(\mathcal{U}^i_{\mathsf{W}}) \neq \emptyset$, and $|\mathcal{A}|$ is polynomial in $|G|$, exponential in $|\psi_1|$, and doubly-exponential in $|\psi_2|, \ldots, |\psi_k|$. Since nonemptiness of NBTs can be checked in quadratic time [26], the upper bound follows. Moreover, when $L(\mathcal{A}) \neq \emptyset$, the algorithm returns a witness to $\mathcal{A}$'s nonemptiness, namely a profile tree that is a solution to the CRS problem.

The construction of $\mathcal{A}$ involves two challenges. First, a naive analysis of the blow-up involved in translating UCTs to NBTs is exponential in the state space of the UCT. In our case, the state space of a UCT $\mathcal{U}^i_{\mathsf{W}}$ is of the form $V \times S$, for some set S that is independent of G. Also, the V-component is updated deterministically: all states sent to the same direction v of the tree agree on their V-element. Consequently, the exponential blow up is only in the S-component, which depends only on $|\psi_i|$. Second, the transformation of UCTs to NBTs that is described in [20] preserves nonemptiness, whereas here we need to preserve nonemptiness of an intersection of automata. As detailed in [17], where we coped with a similar challenge, this can be handled by parameterizing the construction in [20] by a rank (essentially, a bound on the size of transducers that generate trees in the language of the automaton) that corresponds to the size of the intersection.

We continue to the lower bounds, and we show they are valid already in the case $k = 2$. Proving an EXPTIME lower bound in $|\psi_1|$, we describe a reduction from the membership problem for linear-space alternating Turing machines (ATM), defined in the full version. That is, given an ATM M with space complexity $s : \mathbb{N} \rightarrow \mathbb{N}$ and a word w, we construct a 2-player turn-based game $\mathcal{G} = \langle G, \{\psi_1, \psi_2\}\rangle$, such that G and ψ_2 are of a fixed size, ψ_1 is of size linear in $s(|w|)$, and there is a CRS solution in $\mathcal{G}$ iff M accepts w.

Essentially, Player 1 and Player 2 generate a branch in the computation tree of M on w. Player 1 chooses the letters of the current configuration one by one, and chooses, at the end of each existential configuration, the successor configuration to which the branch continues. Player 2, on the other hand, only chooses successor configurations at the end of each universal configuration (see Fig. 1). The objective of Player 1 is to reach an accepting configuration, and the objective of Player 2 is to reach a rejecting configuration.

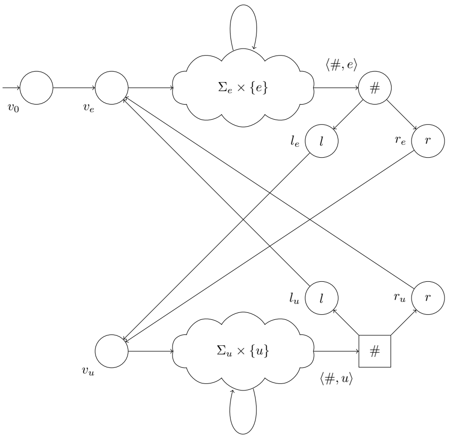


Fig. 1. The game graph G. The circles are vertices controlled by Player 1, and the square is a vertex controlled by Player 2.

We prove that $\mathcal{G}$ has a $\{1\}$-NE that satisfies ψ_1 iff M accepts w. First, if M accepts w, then the profile in which Player 1 follows a strategy that generates the configurations in the accepting computation and chooses the appropriate successors to existential configurations, is a $\{1\}$-NE that satisfies ψ_1. Also, if M rejects w, then Player 2 can choose successors of universal configurations in a way that leads to a rejecting configuration. Thus, there is no $\{1\}$-NE in $\mathcal{G}$ that satisfies ψ_1, as either Player 1 loses by not forming a valid branch, or Player 2 can deviate to a strategy where she wins and Player 1 loses. In the full version, we give the details of the reduction.

Proving a 2EXPTIME lower bound in $|\psi_2|$, we use a reduction from decidability of 2-player zero-sum games, which is 2EXPTIME-hard already for a game with a game graph of a fixed size [2]. Given a 2-player zero-sum game $\mathcal{G} = \langle G, \psi \rangle$, we construct a 2-player game $\mathcal{H} = \langle H, \{\psi_1, \psi_2\} \rangle$ such that the size of H is linear in $|G|$, ψ_1 is of a fixed size, ψ_2 is of size linear in $|\psi|$, and there is a CRS solution in $\mathcal{H}$ iff Player 1 wins $\mathcal{G}$. Essentially, the game graph H contains two copies of G, and a new initial vertex in which Player 2 chooses between proceeding to the first or the second copy.

Note that Player 1 has no influence in that decision. Then, the objective of Player 1 is for the play to be generated in the first copy, and the objective of Player 2 is for the play to be generated in the second copy and for the computation to not satisfy ψ. □

Remark 4. Note that our algorithm finds W-CRS solutions for all $\mathsf{W} \subseteq [k]$, and so it is exponential in k. As shown in [9], rational synthesis is PSPACE in k already for rational synthesis with reachability objectives.

5 The Complexity of Non-Cooperative Rational Synthesis

In this section we study the complexity of NRS. We start with the turn-based setting, and then proceed the concurrent setting.

5.1 Turn-based games

As in the CRS case, we construct a tree automaton that accepts strategy trees that are NRS solutions. Here, however, the trees are labeled not only by a strategy for Player 1, but also by information that certifies that the suggested strategy is indeed a solution. Our construction follows the ideas developed for turn-based games in [9], adding to them a treatment of the LTL objectives (the latter is not too complicated, and our main goal in this section is to set the stage to the concurrent setting, which was left open in [9]). In order to present our solution, we first need some definitions and notations.

Consider a k-player turn-based game $\mathcal{G} = \langle G, \{\psi_i\}_{i \in [k]} \rangle$. Let $G = \langle AP, \{V_i\}_{i \in [k]}, v_0 , E, \tau \rangle$. Recall that for a subset $S \subseteq [k]$ of players, an S*-profile* is a set of strategies, one for each player in S, and that a profile π *extends* an S-profile π' if the players in S use in π their strategies in π'. The outcome of an S-profile π', denoted $\mathsf{Outcome}(\pi')$, is the union of plays that are outcomes of profiles that extend π'. Thus, $\mathsf{Outcome}(\pi') \subseteq V^\omega$ is the set of plays that are possible outcome of the game when the players in S follow their strategies in π'.

Consider a profile $\pi = \langle f_1, \dots, f_k \rangle$ and a prefix $h \in V^*$ of $\mathsf{Outcome}(\pi)$. For a profile $\pi' = \langle f'_1, \dots, f'_k \rangle$, we define the profile $\mathsf{switch}(\pi, \pi', h) = \langle f^h_1, \dots, f^h_k \rangle$ as the profile in which the players first follow π and generate h, and then switch to following π'. Formally, for every $x \in V^*$ and Player $i \in [k]$, if $x \preceq h$, then $f^h_i(x) = f_i(x)$, and if $x = h \cdot y$, then $f^h_i(x) = f'_i(y)$. Note that since the last vertices in x and y coincide, then $\mathsf{switch}(\pi, \pi', h)$ is well defined, in the sense that it returns only allowed actions. Also note that $f^h_i(h) = f'_i(\varepsilon)$, thus, switching to following π', we reset the history of the game so far. The strategies in nodes that are neither a prefix of h nor an extension of h are arbitrary and can follow π.

For $i \in [k] \setminus \{1\}$ and a prefix $h \in V^* \cdot V_i$ of some play in $\mathsf{Outcome}(\{f_1\})$, we say that *Player i wins from h* if there exists a strategy f'_i for Player i such that for every profile $\pi = \langle f_1, \ldots, f_k \rangle$ with $h \preceq \mathsf{Outcome}(\pi)$, we have that $i \in \mathsf{Win}(\mathsf{switch}(\pi, \pi[i \leftarrow f'_i], h))$. That is, Player i wins in every profile in which the players first generate h, and then Player i switches to following f'_i, while the other players adhere to their strategies in the original profile. The strategy f'_i is then called an *h-winning strategy for Player i.*

Since turn-based games always have an NE, an NRS solution in $\mathcal{G}$ is a strategy f_1 for Player 1 such that for every 1-fixed NE $\pi = \langle f_1, \ldots, f_k \rangle$, we have that $1 \in \mathsf{Win}(\pi)$. Equivalently, for every profile $\pi = \langle f_1, \ldots, f_k \rangle$, we have that either $1 \in \mathsf{Win}(\pi)$, or there exists $i \in \mathsf{Lose}(\pi)$ such that $i \in \mathsf{Win}(\pi[i \leftarrow f'_i])$ for some strategy f'_i for Player i. As detailed in [9], this implies that a strategy f_1 for Player 1 is an NRS solution iff for every path ρ in $\mathsf{Outcome}(\{f_1\})$, either $\tau(\rho) \in L(\psi_1)$, or there is $i \in [k] \setminus \{1\}$ such that $\tau(\rho) \notin L(\psi_i)$, and there are a prefix $h \preceq \rho$ and an h-winning strategy f'_i for Player i. We then say that h is a *good deviation point* for Player i, and f'_i is a *good deviation* for Player i.

Our goal is to define a tree automaton that accepts a strategy tree for Player 1 iff it is an NRS solution. The tree automaton should check that every path in $\mathsf{Outcome}(\{f_1\})$ that does not satisfy ψ_1 has a good deviation point for one of the players that lose in it. For that purpose, a strategy f_1 of Player 1 is going to be *certified* by information about deviations: each path in $\mathsf{Outcome}(\{f_1\})$ that does not satisfy ψ_1 is labeled by a player i that loses in the path, a good deviation point for Player i, and a good deviation for Player i. Note that each deviation may handle only a subset of the paths below the good deviation point, and thus a subtree in the certified strategy tree may be labeled by strategies of different players, each deviating at different points.

Formally, a certified strategy tree is a $((V \cup \{\oslash\}) \times [k])$-labeled V-tree $\langle V^*, g \rangle$, where each node is labeled by a pair $\langle v, i \rangle$, where $v \in V \cup \{\oslash\}$ is a *strategy-label*, and $i \in [k]$ is a *player-label.* We use g_s and g_p to refer to the projection of g on the strategy and player components. Each path in the tree that corresponds to a play in $\mathsf{Outcome}(\{f_1\})$ has a suffix all whose nodes are labeled by the same player label. If this label is 1, then the strategy labels describe a strategy of Player 1 and the path should satisfy ψ_1. If this label is $i \in [k] \setminus \{i\}$, then a deviation point of Player i has been encountered, the strategy labels describe a good deviation for Player i, and the path should not satisfy ψ_i. As long as a deviation point has not been encountered, the strategy labels describe a strategy for Player 1 (and so, they are in V in nodes with a direction in V_1, and are $\oslash$ in nodes with a direction not in V_1). Once a deviation point for Player i is encountered (which is indicated by the strategy label being changed from $\oslash$ to a vertex in V in a node with direction V_i), the strategy labels describe a strategy for Player i.

By adjusting Lemma 7 in [9] to the setting with LTL objectives, we get the following.

Theorem 3. *A strategy f_1 for Player 1 is an NRS solution iff there is a certified strategy tree $\langle V^*, g \rangle$ that agrees with f_1. Thus, for every $h \in V^*$ and $v \in V_1$ such that $h \cdot v \in \mathsf{Outcome}(\{f_1\})$, we have that $g_s(h \cdot v) = f_1(h \cdot v)$*

We now define a tree automaton that accepts certified strategy trees, which we then use for solving NRS.

Theorem 4. *We can construct a UCT $\mathcal{U}$ over $((V \cup \{\oslash\}) \times [k])$-labeled V-trees such that $\mathcal{U}$ accepts a $((V \cup \{\oslash\}) \times [k])$-labeled V-tree $\langle V^*, g\rangle$ iff $\langle V^*, g\rangle$ is a certified strategy tree. The size of $\mathcal{U}$ is polynomial in $|G|$ and exponential in $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$.*

Proof. The requirements on a certified strategy tree $\langle V^*, g\rangle$ for Player 1 can be decomposed to the following conditions.

- ($\mathbf{C_1^i}$) For every $i \in [k] \setminus \{1\}$, the subtree of every node $h \in V^* \cdot V_i$ in the tree that is labeled by $V \times [k]$ is labeled by an h-winning strategy for Player i.
- ($\mathbf{C_2}$) The (infinite) suffix of every path in the tree is p-labeled by a single $i \in [k]$.
- ($\mathbf{C_3^i}$) For every $i \in [k] \setminus \{1\}$, every path in the tree with a suffix p-labeled by i has a good deviation point for Player i.
- ($\mathbf{C_4^1}$) Player 1 wins in every path in the tree with suffix p-labeled by 1.
- ($\mathbf{C_4^i}$) for every $i \in [k] \setminus \{1\}$, Player i loses in every path in the tree with suffix p-labeled by i.

In the full version, we describe UCTs that check these conditions and whose intersection is of the desired size. □

Theorem 5. *Solving NRS can be done in time polynomial in $|G|$ and doubly-exponential in $|\psi_1|, \ldots, |\psi_k|$. The problem is 2EXPTIME-hard in each of $|\psi_1|$, $\ldots$, $|\psi_k|$.*

Proof. We start with the upper bound. By Theorems 3 and 4, we can reduce NRS to nonemptiness of a UCT $\mathcal{U}$ over $((V \cup \{\oslash\}) \times [k])$-labeled V-trees of size polynomial in $|G|$ and exponential in $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$. Using considerations similar to these used in the proof of Theorem 2 (in particular, the fact $\mathcal{U}$ is deterministic in its V-element), we can construct from it an NBT $\mathcal{N}$ of size polynomial in $|G|$ and doubly-exponential in $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$ that preserves the nonemptiness of $\mathcal{U}$. Since the nonemptiness problem for NBT can be solved in quadratic time [26], the desired complexity follows.

We continue to the lower bounds, and we show they are valid already in the case $k = 2$. We again use reductions from deciding 2-player zero-sum games. In order to prove 2EXPTIME-hardness in $|\psi_1|$, consider a 2-player zero-sum game $\mathcal{G} = \langle G, \psi\rangle$, for a fixed-size G. We claim that the 2-player game $\mathcal{G}' = \langle G, \{\psi, \mathbf{true}\}\rangle$ is such that G is of a fixed size and that there is an NRS solution in $\mathcal{G}'$ iff Player 1 wins $\mathcal{G}$. Indeed, since the objective of Player 2 is **true**, every profile π in $\mathcal{G}'$ is a 1-fixed NE. So, in order for a strategy f_1 to be an NRS solution, it must satisfy that $1 \in \mathsf{Win}(\langle f_1, f_2\rangle)$, for every strategy f_2 for Player 2. Equivalently, it is a winning strategy for Player 1 in $\mathcal{G}$.

In order to prove 2EXPTIME-hardness in $|\psi_2|$, consider again a 2-player zero-sum game $\mathcal{G} = \langle G, \psi\rangle$, for a fixed-size G. We construct a 2-player game $\mathcal{G}' = \langle H, \{\psi_1, \psi_2\}\rangle$ such that H and ψ_1 are of a fixed size, the size of ψ_2 is linear in $|\psi|$, and there is an NRS solution in $\mathcal{G}'$ iff Player 1 wins $\mathcal{G}$. The game graph H is as in the proof of Theorem 2. Thus, it has an initial vertex from which Player 2 chooses between two copies of G. The states of the first copy are labeled by a fresh atomic proposition p. Then, $\psi_1 = XGp$, and $\psi_2 = X((\psi \wedge Gp) \vee ((\neg\psi) \wedge G\neg p))$. Thus, the objective of Player 1 is for the play to be generated in the first copy, and the objective of Player 2 is either to generate a play in the first copy whose computation satisfies ψ, or to generate a play in the second copy whose computation does not satisfy ψ.

If Player 1 has a winning strategy f_1 in $\mathcal{G}$, then there is an NRS solution f_1' in $\mathcal{G}'$, where f_1' follows f_1 in the copy Player 2 chooses. Indeed, as f_1' guarantees the satisfaction of ψ for all possible behaviors of Player 2, a profile π is a 1-fixed NE only if Player 2 chooses the first copy. If Player 1 loses $\mathcal{G}$, then for every strategy f_1 for Player 1, there is a strategy f_2 for Player 2 such that $\mathsf{Outcome}(\langle f_1, f_2\rangle)$ does not satisfy ψ. So, for every strategy f_1 for Player 1 in $\mathcal{G}'$, we have that there is a 1-fixed NE $\pi = \langle f_1, f_2\rangle$ such that $1 \in \mathsf{Lose}(\pi)$, where f_2 is the strategy that chooses the second copy, and ensures that ψ is not satisfied. Hence, there is no NRS solution in $\mathcal{G}'$. □

5.2 Concurrent games

Consider a k-player concurrent game $\mathcal{G} = \langle G, \{\psi_i\}_{i\in[k]}\rangle$. Let $G = \langle AP, V, v_0, \{A_i\}_{i\in[k]}, \{\kappa_i\}_{i\in[k]}, \delta, \tau\rangle$. As our constructions in this section are loaded with notations, we simplify the setting and assume that there is one set A of actions, available to all players in all vertices. That is, $A_1 = A_2 = \cdots = A_k = A$, and for every $v \in V$ and $i \in [k]$, we have that $\kappa_i(v) = A$. All our constructions and results can be easily extended to the general case.

As in the turn-based setting, we define a UCT that accepts certified strategy trees for Player 1. In the concurrent setting, however, certification is much more complicated. Below we explain the challenges in the concurrent setting and how we overcome them. For $i \in [k] \setminus \{1\}$ and a prefix $h \in V^*$ of some path in $\mathsf{Outcome}(\{f_1\})$, we say that *Player i wins from h* if for every profile $\pi = \langle f_1, \ldots, f_k\rangle$ with $h \preceq \mathsf{Outcome}(\pi)$, we have that there exists a strategy f_i' for Player i such that $i \in \mathsf{Win}(\mathsf{switch}(\pi, \pi[i \leftarrow f_i'], h))$. Thus, Player i wins from h if she has a beneficial deviation to switch to from h, for every profile π with $h \preceq \mathsf{Outcome}(\pi)$. Note that for different profiles, Player i might have different such beneficial deviations. Here, however, the prefix h need not end in V_i (in fact, there is no V_i in the concurrent setting). We say that h is a *winning point* for Player i. Also, we say that $(h, \langle f_1(h), \ldots, f_k(h)\rangle)$ is a *good deviation pair* for Player i iff there exists $a_i' \in A$ such that $h \cdot \delta(h, \langle f_1(h), \ldots, a_i', \ldots, f_k(h)\rangle)$ is a winning point for Player i.

In order to understand better the difference between NRS solutions in the turn-based and concurrent settings, recall that a strategy f_1 for Player 1 is not an NRS solution iff there is a 1-fixed NE $\pi = \langle f_1, \ldots, f_k\rangle$ whose outcome ρ does not satisfy ψ_1. Note that π being a 1-fixed NE means that for every prefix $h\cdot v\cdot u$ of ρ, there exist actions $\langle a_2, \ldots, a_k\rangle \in A^{k-1}$ such that $\delta(v, \langle f_1(h\cdot v), a_2, \ldots, a_k\rangle) = u$ and for every $i \in \mathsf{Lose}(\rho)$, we have that $(h\cdot v, \langle f_1(h\cdot v), a_2, \ldots, a_k\rangle)$ is not a good deviation pair for Player i. In particular, we can choose $a_i = f_i(h \cdot v)$. Hence, if f_1 is an NRS solution, and there exists a path $\rho \in \mathsf{Outcome}(\{f_1\})$ that does not satisfy ψ_1, then there must be a prefix $h \cdot v \cdot u \preceq \rho$ such that for every $\langle a_2, \ldots, a_k \rangle \in A^{k-1}$ with $\delta(v, \langle f_1(h \cdot v), a_2, \ldots, a_k \rangle) = u$, there exists $i \in \mathsf{Lose}(\rho)$ such that $(h \cdot v, \langle f_1(h \cdot v), a_2, \ldots, a_k \rangle)$ is a good deviation point for Player i. We then say that $h \cdot v \cdot u$ is a *good deviation transition for* $\mathsf{Lose}(\rho)$. Thus, while in the turn-based settings it is sufficient to find in every path in which Player 1 loses a good deviation point for one of the players that lose in it, in the concurrent setting the definition of good deviation depends on the transition induced by the specific profile being used, and so we have to consider deviating transitions, and there may be several players in $\mathsf{Lose}(\rho)$ that deviate. Accordingly, in order to certify a strategy for Player 1, we should describe a mapping from every vector of actions to a set of players, along with their deviations.

Another difference between the turn-based setting and the concurrent setting is that only in the first, the existence of some 1-fixed NE is guaranteed [7]. Hence, we have to add to the algorithm such a check (which is in fact easy).

We can now define certified strategy trees for the concurrent setting. Every node in a certified strategy tree is labeled by the following components:

1. An action $a_1 \in A$, which is the strategy for Player 1.
2. A *deviation function* $d : A^{k-1} \to (A \cup \{\perp\})^{k-1}$, which maps a vector of actions of players $2, \ldots, k$ to the set of players that deviate from it, along with their deviations. Specifically, $\langle a_2, \ldots, a_k \rangle \in A^{k-1}$ being mapped to $\langle a'_2, \ldots, a'_k \rangle \in (A \cup \{\perp\})^{k-1}$ indicates that for every $i \in [k] \setminus \{1\}$, if $a'_i \in A$, then a'_i is the deviation for Player i from $\langle a_2, \ldots, a_k \rangle$, and if $a'_i = \perp$ then no deviation from $\langle a_2, \ldots, a_k \rangle$ is specified for Player i. Let $\mathcal{D}$ denote the set of all possible deviation functions.
3. A set $\mathsf{L} \subseteq \{2, \ldots, k\}$ of players, which describes the set of players that lose in a given path and which are therefore expected to have a good deviation transition. That is, if a suffix of a path is labeled by $\emptyset$, then Player 1 should win in this path, and if a suffix of a path is labeled by $\mathsf{L} \neq \emptyset$, then all the players in L lose in this path.
4. A vector of actions $\langle a_2, \ldots, a_k \rangle \in A^{k-1}$, which describes the strategies for the other players in the required 1-fixed NE.

Formally, a certified strategy tree is a $(A \times \mathcal{D} \times 2^{\{2,\ldots,k\}} \times A^{k-1})$-labeled V-tree $\langle V^*, g \rangle$, where each node is labeled by both a *strategy-label* $a_1 \in A$, a *deviation-label* $d \in \mathcal{D}$, a *player-label* $\mathsf{L} \in 2^{\{2,\ldots,k\}}$, and an *NE-label* $\langle a_2, \ldots, a_k \rangle \in A^{k-1}$. We use g_s, g_d, g_p, and g_{NE} to refer to the projection of g on its different components.

For a node $h \cdot v$ that is s-labeled by a_1 and d-labeled by d, a possible successor u of v, and a set of losers L, we say that $h \cdot v \cdot u$ is *marked as a good deviation transition for* L iff the following hold:

1. For every $\langle a_2, \ldots, a_k \rangle \in A^{k-1}$ such that $\delta(v, \langle a_1, a_2, \ldots, a_k \rangle) = u$, there exists $i \in \mathsf{L}$ such that $(d(\langle a_2, \ldots, a_k \rangle))_i \in A$. That is, for every vector of actions $\langle a_2, \ldots, a_k \rangle$ that leads to u, there is $i \in \mathsf{L}$ such that d assigns a deviation for Player i from $\langle a_2, \ldots, a_k \rangle$.

2. For every $i \in \mathsf{L}$, there is a vector of actions $\langle a_2, \ldots, a_k \rangle \in A^{k-1}$ such that $\delta(v, \langle a_1, a_2, \ldots, a_k \rangle) = u$, and $(d(\langle a_2, \ldots, a_k \rangle))_i \in A$. That is, for every $i \in \mathsf{L}$ there exists a vector of actions that leads to u, from which d assigns a deviation for Player i.

Now, an $(A \times \mathcal{D} \times 2^{\{2,\ldots,k\}} \times A^{k-1})$-labeled V-tree $\langle V^*, g \rangle$ is a certified strategy tree iff it satisfies the following conditions:

- $(\mathbf{C_1})$ If $h \cdot v \cdot u \in V^*$ is marked as a good deviation transition for a set of players $\mathsf{L} \subseteq \{2, \ldots, k\}$, $\mathsf{L} \neq \emptyset$, then it is indeed a good deviation transition for L.
- $(\mathbf{C_2})$ Every path ρ has a set L such that ρ is eventually always p-labeled by L.
- $(\mathbf{C_3^L})$ For every $\mathsf{L} \subseteq \{2, \ldots, k\}$, $\mathsf{L} \neq \emptyset$, every path in the tree with a suffix p-labeled by L has a good deviation transition for L.
- $(\mathbf{C_4^1})$ Player 1 wins in every path in the tree with suffix p-labeled by $\emptyset$.
- $(\mathbf{C_4^i})$ For every $i \in [k] \setminus \{1\}$, Player i loses in every path in the tree with suffix p-labeled by L such that $i \in \mathsf{L}$.
- $(\mathbf{C_5})$ The s and NE-labeling of the tree specifies a 1-fixed NE.

Theorem 6. *A strategy f_1 for Player 1 is an NRS solution iff there is a certified strategy tree $\langle V^*, g \rangle$ that agrees with f_1. That is, for every $h \in V^*$, we have that $g_s(h) = f_1(h)$.*

Theorem 7. *We can construct a UCT over $(A \times \mathcal{D} \times 2^{\{2,\ldots,k\}} \times A^{k-1})$-labeled V-trees that accepts a $(A \times \mathcal{D} \times 2^{\{2,\ldots,k\}} \times A^{k-1})$-labeled V-tree $\langle V^*, g \rangle$ iff $\langle V^*, g \rangle$ is a certified strategy tree. The size of the UCT is polynomial in $|G|$ and exponential in $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$.*

Proof. We can construct UCTs for $\mathbf{C_2}$, $\mathbf{C_3^L}$, and $\mathbf{C_4^i}$ that are very similar to the UCTs for $\mathbf{C_2}$, $\mathbf{C_3^i}$ and $\mathbf{C_4^i}$ from the turn-based setting. The UCT for $\mathbf{C_5}$ is similar to the UCT for CRS solutions. In the full version, we describe in detail a UCT that checks the satisfaction of $\mathbf{C_1}$. Then, the conjunction of the above UCTs results in a UCT that accepts certified strategy trees. □

The number of deviation functions $d : A^{k-1} \to (A \cup \{\bot\})^{k-1}$ is exponential in $|A|$. Consequently, the UCT described in Theorem 7 has an exponential alphabet, which causes the NBT generated in [20] to have exponentially many transitions, making its nonemptiness problem exponential. In order to overcome this problem, we introduce *vertical annotation of certified strategy trees*, which essentially replace a node labeled by $d \in \mathcal{D}$ by a sequence of nodes, labeled by a smaller alphabet.

Explaining our vertical annotation, we find it clearer to go back to a detailed description of the actions of the different players, thus refer to A_i and κ_i rather than assuming they are all equal to A. For every vertex $v \in V$ and an action $a_1 \in \kappa_1(v)$ for Player 1, we denote by T_{v,a_1} the set of possible vectors of actions from v, given Player 1 chose a_1. That is, $T_{v,a_1} = \{a_1\} \times \kappa_2(v) \times \cdots \times \kappa_k(v)$. We

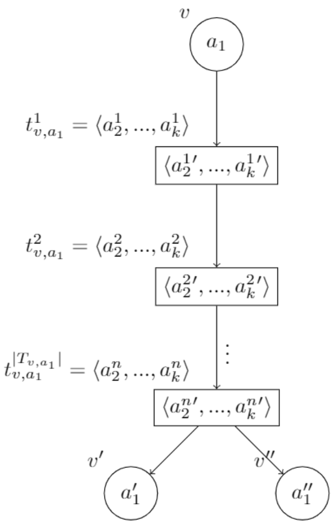

v
a_1
$t^1_{v,a_1} = \langle a^1_2, ..., a^1_k \rangle$
$\langle a^{1\prime}_2, ..., a^{1\prime}_k \rangle$
$t^2_{v,a_1} = \langle a^2_2, ..., a^2_k \rangle$
$\langle a^{2\prime}_2, ..., a^{2\prime}_k \rangle$
$\vdots$
$t^{|T_{v,a_1}|}_{v,a_1} = \langle a^n_2, ..., a^n_k \rangle$
$\langle a^{n\prime}_2, ..., a^{n\prime}_k \rangle$
v'
v''
a'_1
a''_1

Fig. 2. A vertically certified strategy tree. Information about deviations from a given vector of actions is stored in an intermediate node that corresponds to that vector. There is no good deviation transition starting from v.

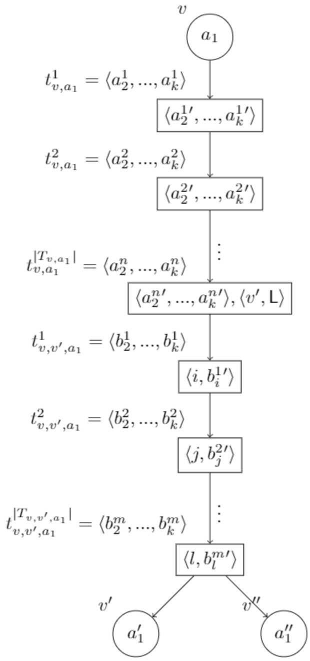

v
a_1
$t^1_{v,a_1} = \langle a^1_2, ..., a^1_k \rangle$
$\langle a^{1\prime}_2, ..., a^{1\prime}_k \rangle$
$t^2_{v,a_1} = \langle a^2_2, ..., a^2_k \rangle$
$\langle a^{2\prime}_2, ..., a^{2\prime}_k \rangle$
$\vdots$
$t^{|T_{v,a_1}|}_{v,a_1} = \langle a^n_2, ..., a^n_k \rangle$
$\langle a^{n\prime}_2, ..., a^{n\prime}_k \rangle, \langle v', \mathsf{L} \rangle$
$t^1_{v,v',a_1} = \langle b^1_2, ..., b^1_k \rangle$
$\langle i, b^{1\prime}_i \rangle$
$t^2_{v,v',a_1} = \langle b^2_2, ..., b^2_k \rangle$
$\langle j, b^{2\prime}_j \rangle$
$\vdots$
$t^{|T_{v,v',a_1}|}_{v,v',a_1} = \langle b^m_2, ..., b^m_k \rangle$
$\langle l, b^{m\prime}_l \rangle$
v'
v''
a'_1
a''_1

Fig. 3. $h \cdot v \cdot v'$ is a good deviation transition for L, where the root v is reachable via history h.

order the vectors in T_{v,a_1} arbitrarily, and, for $1 \le i \le |T_{v,a_1}|$, denote by t^i_{v,a_1} the i-th item in T_{v,a_1}. We also denote by $t^i_{v,a_1}[j \leftarrow a'_j]$ the vector of actions obtained from t^i_{v,a_1} by replacing the action for Player j by a'_j.

For a given transition from v to v', let T_{v,v',a_1} be the restriction of T_{v,a_1} to vectors $\langle a_1, a_2, \ldots, a_k \rangle$ such that $\delta(v, \langle a_1, a_2, \ldots, a_k \rangle) = v'$, and let t^i_{v,v',a_1} denote the i-th item in T_{v,v',a_1}. Also, let $T = \bigcup_{v \in V} \bigcup_{a_1 \in \kappa_1(v)} T_{v,a_1}$, and $\Sigma = A_1 \cup ((A_2 \times \cdots \times A_k) \cup \bigcup_{j \in \mathsf{L}}(\{j\} \times A_j)) \times (\emptyset \cup (V \times 2^{\{2,\ldots,k\}}))$. We also need an additional $2^{\{2,\ldots,k\}}$ component, for annotating the set of losers in each path in the tree, but we omit it for now, as it is not relevant for the vertical annotations.

A certified strategy tree is now a Σ-labeled $(V \cup T)$-tree, where nodes with direction in V are labeled by the strategy for Player 1, and nodes with direction in T are labeled with deviation information. Consider a node that corresponds to a vertex v with history h and is labeled by a_1. Following vertically there are nodes corresponding to T_{v,a_1}, each labeled by a vector of actions (see Fig. 2). This information is for verifying good deviation transitions. That is, after a good deviation transition is announced, we need to verify that the involved players indeed have appropriate beneficial deviations. The last node in the chain, which corresponds to $t^{|T_{v,a_1}|}_{v,a_1}$, is either not labeled, or labeled by a vertex v' and a set L of losers. If it is not labeled, it means there are no good deviation transitions from v, in which case we continue to the nodes corresponding to the possible successors of v. If it is labeled by $\langle v', \mathsf{L}\rangle$, it means that $h \cdot v \cdot v'$ is a good deviation transition for L (see Fig. 3). Hence, the following nodes correspond to T_{v,v',a_1}, each labeled by a single deviation, followed by a chain of good deviation transitions, using the appropriate $V \times 2^{\{2,\ldots,k\}}$ annotations, or by nodes corresponding to successor vertices. In the full version, we describe formally a UCT for verifying good deviation transitions in vertically annotated certified strategy trees.

Theorem 8. *Solving NRS in the concurrent setting can be done in time polynomial in* $|G|$ *and doubly-exponential in* $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$*. The problem is 2EXPTIME -hard in each of* $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$*.*

Proof. We start with the upper bound. We can easily modify the UCTs for $\mathbf{C_2}$, $\mathbf{C_3^{\mathsf{L}}}$, $\mathbf{C_4^{i}}$, and $\mathbf{C_5}$ from the proof of Theorem 7 to accommodate the vertical annotation. With conjunction with the UCT for verifying good deviation transitions, described in the full version, we have a UCT $\mathcal{U}$ over Σ-labeled $(V \cup T)$-trees such that $\mathcal{U}$ accepts a Σ-labeled $(V \cup T)$-tree $\langle V^*, g\rangle$ iff $\langle V^*, g\rangle$ is a vertically annotated certified strategy tree. By Theorem 6, there is an NRS solution in $\mathcal{G}$ iff $\mathcal{U}$ is not empty. The size of $\mathcal{U}$ is polynomial in $|G|$ and exponential in $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$, and its alphabet is polynomial in G. Also, $\mathcal{U}$ is deterministic in its V-element. Hence, as detailed in the proof of Theorem 2, we can construct from $\mathcal{U}$ an NBT $\mathcal{N}$ of size polynomial in $|G|$ and doubly-exponential in $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$ such that $L(\mathcal{U}) \neq \emptyset$ iff $L(\mathcal{N}) \neq \emptyset$. Since the nonemptiness problem for NBT can be solved in quadratic time [26], the desired complexity follows.

Finally, as turn-based games are a special case of concurrent ones, the lower bound from Theorem 5 applies here. □

5.3 General rational synthesis

Consider a k-player game $\mathcal{G} = \langle G, \{\psi_i\}_{i\in[k]}\rangle$. Recall that the problem of rational synthesis is to return a $(\{1\} \cup \mathsf{CU})$-profile π' such that there is a 1-fixed NE that extends π', and for every 1-fixed NE π that extends π', we have that $1 \in \mathsf{Win}(\pi)$.

A $(\{1\} \cup \mathsf{CU})$-profile π' is an RS-solution iff for every path ρ such that $\rho \not\models \psi_1$, there is no 1-fixed NE π that extends π' and $\mathsf{Outcome}(\pi) = \rho$. It is easy to see that we can define certified $(\{1\} \cup \mathsf{CU})$-profile trees in a similar way we defined certified strategy trees for Player 1, inducing an algorithm of the same complexity for checking the existence of a certified $(\{1\} \cup \mathsf{CU})$-profile tree. Also, as NRS is a special case of RS, the NRS lower bound provide a lower bound for RS. Hence, we can conclude with the following.

Theorem 9. *Solving RS can be done in time polynomial in* $|G|$ *and doubly-exponential in* $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$*. The problem is 2EXPTIME-hard in each of* $|\psi_1|, |\psi_2|, \ldots, |\psi_k|$*.*

References

1. Almagor, S., Kupferman, O., Perelli, G.: Synthesis of controllable Nash equilibria in quantitative objective game. In: Proc. 27th Int. Joint Conf. on Artificial Intelligence. pp. 35–41 (2018)
2. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM **49**(5), 672–713 (2002)
3. Anshelevich, E., Dasgupta, A., Kleinberg, J., Tardos, E., Wexler, T., Roughgarden, T.: The price of stability for network design with fair cost allocation. In: Proc. 45th IEEE Symp. on Foundations of Computer Science. pp. 295–304. IEEE Computer Society (2004)
4. Bouyer-Decitre, P., Kupferman, O., Markey, N., Maubert, B., Murano, A., Perelli, G.: Reasoning about quality and fuzziness of strategic behaviours. In: Proc. 28th Int. Joint Conf. on Artificial Intelligence. pp. 1588–1594 (2019)
5. Chatterjee, K., Henzinger, T., Jobstmann, B.: Environment assumptions for synthesis. In: Proc. 19th Int. Conf. on Concurrency Theory. Lecture Notes in Computer Science, vol. 5201, pp. 147–161. Springer (2008)
6. Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. In: Proc. 18th Int. Conf. on Concurrency Theory. pp. 59–73 (2007)
7. Chatterjee, K., Majumdar, R., Jurdzinski, M.: On Nash equilibria in stochastic games. In: Proc. 13th Annual Conf. of the European Association for Computer Science Logic. Lecture Notes in Computer Science, vol. 3210, pp. 26–40. Springer (2004)
8. Church, A.: Logic, arithmetics, and automata. In: Proc. Int. Congress of Mathematicians, 1962. pp. 23–35. Institut Mittag-Leffler (1963)
9. Condurache, R., Filiot, E., Gentilini, R., Raskin, J.F.: The complexity of rational synthesis. In: Proc. 43th Int. Colloq. on Automata, Languages, and Programming. LIPIcs, vol. 55, pp. 121:1–121:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
10. Emerson, E., Jutla, C.: The complexity of tree automata and logics of programs. In: Proc. 29th IEEE Symp. on Foundations of Computer Science. pp. 328–337 (1988)
11. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Proc. 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 6015, pp. 190–204. Springer (2010)
12. Gutierrez, J., Najib, M., Perelli, G., Wooldridge, M.: On computational tractability for rational verification. In: Proc. 28th Int. Joint Conf. on Artificial Intelligence. pp. 329–335. International Joint Conferences on Artificial Intelligence Organization (2019)
13. Gutierrez, J., Perelli, G., Wooldridge, M.J.: Imperfect information in reactive modules games. Inf. Comput. **261**, 650–675 (2018)
14. Kupferman, O., Lustig, Y., Vardi, M., Yannakakis, M.: Temporal synthesis for bounded systems and environments. In: Proc. 28th Symp. on Theoretical Aspects of Computer Science. pp. 615–626 (2011)
15. Kupferman, O., Perelli, G., Vardi, M.: Synthesis with rational environments. Annals of Mathematics and Artificial Intelligence **78**(1), 3–20 (2016)
16. Kupferman, O., Piterman, N.: Lower bounds on witnesses for nonemptiness of universal co-Büchi automata. In: Proc. 12th Int. Conf. on Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 5504, pp. 182–196. Springer (2009)

17. Kupferman, O., Shenwald, N.: Perspective games with notifications. In: Proc. 40th Conf. on Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 182, pp. 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
18. Kupferman, O., Shenwald, N.: Perspective multi-player games. In: Proc. 36th IEEE Symp. on Logic in Computer Science (2021)
19. Kupferman, O., Vardi, G.: Perspective games. In: Proc. 34th IEEE Symp. on Logic in Computer Science. pp. 1 – 13 (2019)
20. Kupferman, O., Vardi, M.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science. pp. 531–540 (2005)
21. Nash, J.: Equilibrium points in n-person games. In: Proceedings of the National Academy of Sciences of the United States of America (1950)
22. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.: Algorithmic Game Theory. Cambridge University Press (2007)
23. Papadimitriou, C.H.: Algorithms, games, and the internet. In: Proc. 33rd ACM Symp. on Theory of Computing. pp. 749–753 (2001)
24. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science **13**, 45–60 (1981)
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages. pp. 179–190 (1989)
26. Vardi, M., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and Systems Science **32**(2), 182–221 (1986)
27. Wooldridge, M., Gutierrez, J., Harrenstein, P., Marchioni, E., Perelli, G., Toumi, A.: Rational verification: From model checking to equilibrium checking. In: Proc. of 30th National Conf. on Artificial Intelligence. pp. 4184–4190 (2016)

Synthesis of Compact Strategies for Coordination Programs

Kedar S. Namjoshi[1](✉) and Nisarg Patel[2]

[1] Nokia Bell Labs, Murray Hill, NJ 07974, USA
kedar.namjoshi@nokia-bell-labs.com
[2] New York University, New York, NY 10001, USA nisarg@nyu.edu

Abstract. In multi-agent settings, such as IoT and robotics, it is necessary to coordinate the actions of independent agents to achieve a joint behavior. While it is often easy to specify the desired behavior, programming the necessary coordination can be difficult. This makes coordination an attractive target for automated program synthesis; however, current methods may produce strategies that issue useless actions. This paper develops theory and methods to synthesize coordination strategies that are guaranteed not to initiate unnecessary actions. We refer to such strategies as being "compact." We formalize the intuitive notion of compactness, show that existing methods do not guarantee compactness, and propose a solution. The solution transforms a given temporal logic specification using automata-theoretic constructions to incorporate a notion of minimality. The central result is that the winning strategies for the transformed specification are precisely the compact strategies for the original. One can therefore apply known synthesis methods to produce compact strategies. We report on prototype implementations that synthesize compact strategies for temporal logic specifications and for specifications of multi-robot coordination.

1 Introduction

Imagine a future home where devices are network-controllable and the control program is synthesized from requirements. Suppose that the homeowner asks for the living-room lights to be turned on when it gets dark. To meet this requirement, a control program must necessarily coordinate the on/off state of the lights with readings from an illumination sensor.

This specification may be expressed more precisely in linear-time temporal logic (LTL) as $\mathsf{G}(\text{dark} \;\Rightarrow\; \mathsf{X}\,\text{light-on})$.[3] Here "dark" is a proposition that represents a reading from the sensor, and is therefore an input to the control program, while "light-on" is a proposition that represents an action, and is therefore an output of the control program. Abstracting this formula to the shape $\mathsf{G}(a \;\Rightarrow\; \mathsf{X}\,b)$, the left half of Figure 1 shows the smallest state machine that

[3] G and X are, respectively, the temporal always and next-time operators. Actions are assumed instantaneous for simplicity.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 46–63, 2022.
https://doi.org/10.1007/978-3-030-99524-9_3

meets this specification. It represents a control program that entirely ignores the sensor input and leaves the lights on all day! This strategy is clearly undesirable, although technically it does meet the specification. The machine on the right represents the "commonsense" controller that keeps the lights on only as long as the sensor indicates that it is dark. The two machines are equally valid from the viewpoint of correctness. How then should we distinguish them? And how can a synthesis method avoid generating undesirable solutions? Those are the questions addressed in this paper.

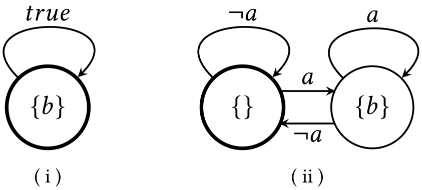


Fig. 1. Non-compact (left) and compact (right) machines for $\mathsf{G}(a \Rightarrow \mathsf{X}\, b)$. The initial state is indicated by a thick border. Output actions are listed at each state; input conditions are placed on the edges.

We suggest that the crucial distinguishing factor is that the left-hand machine invokes actions that are not essential to satisfying the property. For instance, if the input a is false now, there is no need to invoke action b in the next step. If input a remains false, there is no need to invoke action b at all. It is vital to avoid useless actions in the domains of IoT and robotics, where agents interact with the physical world: there is no need to switch on a toaster when only watering the lawn is asked for. Indeed, switching on the toaster unexpectedly may have dangerous side effects. A reader may easily imagine other similar situations.

We refer to the policy of avoiding unnecessary actions as *compactness*. Strategies that satisfy this property while meeting the specification are called *compact*. An immediate question is whether compactness is ensured by standard synthesis methods. Unfortunately, the answer is 'no.' Bounded synthesis [35,20], for instance, will produce the smallest satisfying Mealy or Moore machine; in this setting, the solution of Figure 1(i). We have validated this experimentally with the tool BoSy [19]. Quantitative synthesis (cf. [6]) finds solutions that are worst-case optimal, i.e., programs where the maximum cost, over all input sequences, is the lowest possible. (Dually, programs where the worst-case reward is the highest possible.) Letting each action invocation have unit cost, a quantitative method cannot distinguish between the solutions shown, as both have the same maximum cost for the input where a is always true. We make this analysis precise subsequently, and show that average-case optimality also does not always distinguish compact from non-compact solutions. We have validated this experimentally with the tool QUASY [13]. Hence, compactness cannot be defined in quantitative terms: the synthesis of compact strategies requires new methods.

At its core, the issue of compactness is a variation of the well-known *frame problem* in logic-based AI [29]. The natural way to express the example requirement is as $\mathsf{G}(a \Rightarrow \mathsf{X}\,b)$. However, the semantics of temporal logic allows many satisfying interpretations; among those is the undesirable one of Figure 1(i). This tension between the freedom of interpretation allowed in logic and the naturalness of a specification is at the heart of the frame problem. One approach to achieving compactness is therefore to write a tighter specification, which permits fewer interpretations; e.g., to write the stronger assertion $\mathsf{G}(a \equiv \mathsf{X}\,b)$. But this is not a natural choice. Moreover, reworking a specification by hand to rule out interpretations with unnecessary actions is difficult as the process is not compositional: i.e., one cannot rework portions of a specification separately. The specification transformation defined here performs such a tightening automatically, using automata-theoretic constructions.

The motivating application of compactness is to the synthesis of centralized coordination programs. As formalized in [3], in a coordination problem, a group of independent agents, denoted $A_1, \ldots, A_n$, are guided by an additional synthesized agent, C, so that their joint behavior meets a temporal specification φ. That work describes a specification transformation from φ to φ' that incorporates asynchronous agent behavior and other constraints. This transformation, however, does not guarantee compactness. We take the transformed problem as the starting point for our investigations, and consider the more general question of how to generate a compact solution for a given temporal specification.

We begin by proposing a mathematical definition of the compactness property. Generalizing from the example, one can consider a strategy to be compact if for each input sequence, the sequence of actions produced as output (1) meets the specification and (2) cannot be further improved. We formalize the second notion as minimality with respect to a supplied "better than" preference relation between two output sequences. This formulation is closely related to formalizations of commonsense reasoning, in particular the notion of circumscription introduced by McCarthy in [28].

For coordination problems, a natural preference relation is based on the subset ordering on sets of actions. We say that sequence y is better than sequence x if (1) in each step, the actions issued in y are a subset of the actions issued by x and (2) for at least one step, the actions in y are a strict subset of the actions in x. The smallest compact strategy for $\mathsf{G}(a \Rightarrow \mathsf{X}\,b)$ under this preference relation issues action b precisely when input a is true at the prior step. Otherwise, there is a point where a is false but b is issued at the next step. Removing this occurrence of b produces a better sequence that also satisfies the property. This is precisely the strategy defined by the machine in Fig. 1(ii). Alternative preference relations may order sets of actions by size, or order sequences of actions by the substring relation. One may also compare infinite action sequences by cost (limit average or discounted sum) using comparator automata [2]. The choice of preference relation is driven by the application domain. To accommodate various options, compactness is parameterized by the preference relation.

Technically, a temporal specification φ can be viewed as a language L, a set of infinite words over a joint input-output alphabet. For a preference relation $\prec$ over infinite words, it is natural to formulate the language $\min(L, \prec)$ that contains only the *minimal words* in L with respect to the preference relation. The central theoretical result in this paper is that there is a compact strategy satisfying L if, and only if, there is a strategy satisfying $\min(L, \prec)$. This theorem reduces the question of synthesizing compact strategies to a standard synthesis question, making it possible to use existing synthesis algorithms to produce compact strategies. We give sufficient conditions under which $\min(L, \prec)$ is regular when L is a regular language, and show how to effectively construct a finite automaton for the minimal language and for its complement, from either an automaton or an LTL formula for L. The constructed automata can also be used to model-check whether a given control program defines a compact strategy. Moreover, the transformation makes it possible to modularly apply quantitative or other criteria for synthesis from $\min(L, \prec)$; for instance, to synthesize compact strategies that minimize program size or worst-case execution time.

We have implemented these constructions and used them to synthesize compact strategies for LTL specifications and for a class of specifications that arise in multi-robot coordination. Experiments show that compact strategies exist for many specifications and can be effectively computed, albeit with some added overhead. We also experiment with approximation methods which are simpler and avoid potential worst-case exponential blowups in the general construction.

In our view, the main contributions of this work are in bringing attention to the need for compactness in program synthesis; showing its independence from existing criteria; giving a precise formulation in terms of minimality; and in designing and implementing algorithms to synthesize compact strategies.

2 Background

Automata A finite automaton is a tuple $(Q, \Sigma, \hat{Q}, \delta, F)$ where Q is a set of *states*; Σ is a set of *letters*, an *alphabet*; $\hat{Q}$ is a non-empty set of *initial* states; $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*; and F is a non-empty set of *final* states.

A *word* over Σ is a (possibly empty) sequence of letters from Σ. For a word w, its *length* $|w|$ is the number of letters in w if w is finite and ω if w is infinite. We assume the standard definition of a *run* of the automaton on a word. If w is finite, a run on w is *accepting* if the last state of the run is in F; if w is infinite, a run is accepting by the Büchi condition if a state in F occurs on the run infinitely often. The *language* of an automaton is the set of words for which there exists an accepting run. One typically distinguishes between the finite-word language and the infinite-word language of an automaton. An automaton is *deterministic* if there is exactly one initial state and for every q and a, there is at most one q' such that (q, a, q') is in δ.

We use the standard abbreviations DFA, NFA and NBA for a deterministic automaton, a nondeterministic automaton over finite words, and a nondeterministic Büchi automaton over infinite words respectively.

LTL Linear Temporal Logic (LTL) is a logic defined over a set of *atomic propositions*, *AP*. The logic has the following minimal grammar, where $p \in AP$:

$$f := p \ \mid \ f_1 \wedge f_2 \ \mid \ \neg f_1 \ \mid \ \mathsf{X}\, f_1 \ \mid \ f_1 \,\mathsf{U}\, f_2$$

The satisfaction relation is defined over infinite words where each letter is a subset of AP. It has the form $w, i \models f$ for a word w and a natural number i, and is given by structural induction on formulas. We omit the standard definition. The language of a formula is the set of words that satisfy it. Standard constructions compile an LTL formula to an NBA that accepts the same language (cf. [18,39]), possibly incurring an exponential blowup.

Programs as Transition Systems A program is represented by its state transition system. This is a Moore machine, defined as a tuple $(S, \hat{S}, I, O, R, o)$ where S is a set of *states*, $\hat{S}$ is a non-empty set of *initial* states; I is a set of *input* values; O is a set of *output* values; $R \subseteq S \times I \times S$ is the *transition relation*, which must be total on I; and $o : S \to O$ is the *output mapping.* An *execution* of this system is an unbounded alternating sequence of states and inputs, and takes the form $s_0, a_0, s_1, a_1, \ldots$, such that for each i, the triple (s_i, a_i, s_{i+1}) is in the transition relation. A *computation* is an execution of this form where s_0 is an initial state.

Input-Output Words An input-output word (i/o word for short) is a pair of sequences (a, b) where a is a sequence of inputs, b is a sequence of outputs, and $|b| = 1 + |a|$. The input-output word induced by a program execution $s_0, a_0, s_1, a_1, \ldots$ is the pair (a, b) with $b = o(s_0), o(s_1), \ldots$. We sometimes write an i/o word in the linear format $b_0, a_0, b_1, a_1, \ldots$ for clarity. It is also common (cf. [32]) to view an infinite i/o word (a, b) in the "zipped" form $a \bowtie b = (a_0, b_0), (a_1, b_1), \ldots$. For a temporal property φ defined over input and output predicates and program M, the program M satisfies φ, written $M \models \varphi$, if the zipped input-output word of every computation of M satisfies φ. Each atomic proposition is a function in $I \times O \to \mathsf{Bool}$; an i/o pair (a, b) induces the set of propositions $\{p \mid p(a, b)\}$.

Games and Strategies A *strategy* is a function from finite sequences of inputs to outputs, represented as $\sigma : I^* \to O$. For an infinite input sequence $a = a_0, a_1, \ldots$, the strategy σ induces the infinite output sequence denoted $\sigma(a)$, given by $\sigma(\epsilon), \sigma(a_0), \sigma(a_0, a_1), \ldots$. A *play* for input a is the i/o word $(a, b = \sigma(a))$. We sometimes abuse this notation and use $\sigma(a)$ to refer to the play induced by a. A play is *winning* for a temporal property φ if it satisfies this property when viewed as a zipped i/o word. A strategy σ is *winning* for φ if for every input a, the play on a is winning for φ.

The *realizability* question is: given a property φ, determine whether there *exists* a program satisfying φ. The *synthesis* question is: given a property φ that is realizable, *construct* a program that satisfies φ. A strategy σ induces a *deterministic* program with an infinite state space, denoted $P(\sigma) = (S, \hat{S}, I, O, R, o)$. The state space S is the set of finite input sequences I^*, the initial state is ϵ, the output label for state x is $\sigma(x)$ and the transition relation R is given by

$\{(x, a, xa) \mid x \in I^*, a \in I\}$. This is in fact an infinite complete tree over I (sometimes called a "fulltree") where each node is labeled by an output value. A labeled fulltree, in turn, corresponds to a strategy and a deterministic program.

Synthesis Methods for Temporal Properties There is an extensive literature on methods to synthesize programs from LTL specifications (cf. [34,32,31,26] and tools that implement various algorithms (cf. [33,8,17,27,35,20]), all based on the conversion from LTL formulas to equivalent automata.

The classical approach to realizability of temporal properties (which we only sketch here, cf. [34,32]) is via the connection between programs, strategies, and labeled fulltrees. If a property φ is realizable, there is a deterministic program M satisfying φ. This program may also be seen as a strategy and a fulltree. From a deterministic word automaton with the same language as φ, one constructs a tree automaton that accepts precisely the fulltrees that satisfy φ. Now φ is realizable if and only if the language of this tree automaton is non-empty. For properties in LTL, this procedure can be carried out in 2EXPTIME in the length of the formula φ; the problem is 2EXPTIME-complete [32]. A winning strategy can be extracted as a finite state, deterministic reactive program from the tree automaton, thanks to the finite-model property of temporal logic. This approach is implemented in the tool Strix [30].

Two other approaches have been developed. One is to limit the logic: the GR(1) fragment expresses many useful properties, has a lower complexity (DEXPTIME), and can be implemented easily using symbolic (BDD-based) methods [31]. This is implemented in several tools [33,8,17,27]. The *bounded synthesis* method applies to full LTL and is iterative in nature. By placing bounds on the size of the intended program and the ranking argument for formula satisfaction, one obtains a simpler safety game, which can be solved using symbolic methods [26,35,20]. The approach is implemented in [11,19].

We use two of the approaches described above in this work. The classical approach is used to determine compact realizability of an arbitrary LTL formula, while GR(1) approach is used in the multi-robot setting.

3 Compactness

We formulate compactness for temporal specifications, investigate its properties, and show how to synthesize a compact strategy through a specification transformation. We consider specifications on infinite words for simplicity and to match the semantics of temporal logic.

A relation $\prec_O$ over the set of infinite output words is a *preference* relation if its transitive closure $\prec_O^+$ is irreflexive. We informally say that word b is better than b' if $b \prec_O^+ b'$ holds. As the transitive closure is irreflexive, it is not possible for a word to be better than itself, matching intuition. This relation is extended to input-output words as follows. An i/o word (a, b) is better than an i/o word (a', b') if (1) the input sequences a and a' are identical, and (2) $b \prec_O^+ b'$. The first condition ensures that comparable words have the same input sequence, which is important as we are ultimately interested in the i/o words that are generated as plays of strategies.

Definition 1 (Compact Strategy). *A strategy σ is* compact *for an i/o language L if (1) σ is a winning strategy for L and (2) for every input sequence a, there is no i/o word (a, b') such that (a, b') satisfies L and (a, b') is better than the i/o word $(a, b = \sigma(a))$ that is produced as the play of σ on input a.*

The first condition ensures that σ is a valid strategy for L; the second that a compact strategy produces the "best possible" output for each input. We say that a language L is *compactly realizable* if it has a compact strategy.

Theorem 1. *A language L is realizable if it is compactly realizable. The converse does not hold.*

Proof. From right-to-left, consider a compact strategy σ for L. From the definition, σ is a winning strategy for L, hence L is realizable.

The converse does not hold. Let the input set $I = \{0, 1\}$ and the output set $O = \{c, d\}$ with the output preference ordering $c < d$ extended point-wise to output words. Let the specification L consist of sequences of the form $c(0c)^\omega$ and $d(\{0, 1\}d)^\omega$. This is realizable. No winning strategy can produce c on ϵ as there can be no win on input 1^ω. The single winning strategy produces d on every input sequence, including ϵ. But this strategy is not compact: for input 0^ω it generates $d(0d)^\omega$, but there is the better word $c(0c)^\omega$ in L. □

Standard realizability is monotone: if $L' \subseteq L$ and program M satisfies L', then M also satisfies L. However, compact realizability is neither monotone nor anti-monotone (proof in the full version). As is the case with deduction systems for commonsense reasoning (cf. [37]), non-monotonicity is a consequence of the formulation in terms of minimality.

The simple example from the Introduction is easily extended to a collection of N "if-condition-then-action" requirements. The IFTTT service (https://ifttt.com) or Apple Shortcuts implement these operationally, using an event-driven rule engine. However, from the viewpoint of temporal logic and synthesis, the results can be unexpected, as we have seen. The N requirements in LTL have the shape $(\bigwedge i : \mathsf{G}(a(i) \;\Rightarrow\; \mathsf{X}\, b(i)))$. The smallest model is one with a single state, issuing all the b actions unconditionally. This is clearly unintended. The intended model, which is compact, has 2^N states, one for each subset of the b actions. Thus, the gap between the smallest non-compact and compact models can be exponential in the length of the specification.

We now show the main theorem that links compact and standard realizability through a specification transformation.

Definition 2 (minimal language). *For a language L over alphabet Σ and a preference relation $\prec$ on Σ-words, the minimal elements of L form the language*

$$\min(L, \prec) = \{x \mid x \in L \wedge \neg(\exists y : y \in L \wedge y \prec^+ x)\}$$

I.e., a word x is in $\min(L, \prec)$ if it belongs to L and there is no word y in L that is transitively better than x.

Theorem 2. *Language L is compactly realizable if and only if* $\min(L, \prec)$ *is realizable.*

Proof. (Left-to-right) Let σ be a strategy that compactly realizes L. Consider any input sequence a. The output $b = \sigma(a)$ produced by the strategy is such that there is no word in L that is better than (a, b), by the definition of compactness. Hence, (a, b) is in $\min(L, \prec)$. As this holds for each input sequence, σ is a winning strategy for $\min(L, \prec)$.

(Right-to-left) Let σ be a winning strategy for $\min(L, \prec)$. For any input sequence a and its corresponding output $b = \sigma(a)$, the word $x = (a, b)$ must satisfy $\min(L, \prec)$. By the definition of min, we have that (1) x also satisfies L. Moreover, (2) there is no i/o word y that is better than x and also satisfies L. From (1) and (2), σ is a compact strategy for L. □

3.1 Effective Minimality Constructions for LTL

Theorem 2 implies that one can reduce compact realizability to standard realizability. Given a temporal specification φ, we transform its language $\mathcal{L}(\varphi)$ to the language $\mathcal{C}(\varphi) = \min(\mathcal{L}(\varphi), \prec)$. Starting from an LTL formula f, we give two constructions: one for the minimal language $\mathcal{C}(f)$, the other for its complement. The constructions assume that the relation $\prec^+$ can be expressed as an NBA, which is the case for the preference order defined in the Introduction.

The first construction directly follows Definition 2. The left-hand term ($x \in \mathcal{L}(f)$) is fulfilled by the standard conversion from LTL formula f to an NBA $\mathcal{A}_f$. For the right-hand term, we use the same NBA $\mathcal{A}_f$, now re-defined over y, for the $y \in \mathcal{L}(f)$ term; intersect this with the NBA for $\prec^+$; then project onto x and complement to obtain an NBA for the right-hand conjunct. The intersection of these two NBAs provides an NBA for $\mathcal{C}(f)$. These steps may result a worst-case double exponential blowup in the size of f: the first exponential is in the construction of $\mathcal{A}_f$; the second is in the complementation step. A similar construction applies if the specification is given directly as an NBA.

The second construction produces an NBA for the *complement* of the minimal language, with "only" a worst-case *single* exponential blowup. The complement of $\mathcal{C}(f)$ is (from the definition) $\{x \mid x \notin \mathcal{L}(f) \vee (\exists y : y \in \mathcal{L}(f) \wedge y \prec^+ x)\}$. For an LTL formula f, one constructs NBAs $\mathcal{A}_f$ and $\mathcal{A}_{\neg f}$ for the LTL formulas f and $\neg f$, respectively. An NBA for $(\exists y : y \in \mathcal{L}(f) \wedge y \prec^+ x)$ is obtained as in the first construction by omitting the final complementation step. The union of this NBA with the NBA for $\mathcal{A}_{\neg f}$ gives an NBA for the complement of $\mathcal{C}(f)$.

The NBA for the complement of $\mathcal{C}(f)$ can be used to model-check whether a given strategy is compact. It can also be used to synthesize machines using bounded synthesis, which requires an NBA for the complement of the specification property. The worst-case blowups are unavoidable: that follows from a lower-bound result by Birget [5] and a simpler but less general result of ours, discussed in the full version of this paper.

3.2 Relationship to Quantitative Synthesis

The formulation of compactness is in terms of a *qualitative* notion of minimality. A natural question is the relationship to methods for synthesis with *quantitative* objectives; in particular, methods for producing programs with optimal worst-case or average-case behavior [6,13]. Expanding on the argument in the Introduction, we establish that worst-case optimality cannot always distinguish between compact and non-compact solutions to a given specification.

In quantitative formulations, the synthesis game is formulated so that each transition has an associated reward. The reward of an infinite computation is defined using standard cumulative metrics such as mean-payoff (the limit of average rewards over successively longer prefixes) or discounted sum (the sum of rewards over the computation discounted geometrically, i.e., the k'th reward contributes a factor d^k, where $d \in (0, 1)$ is the discount factor). The objective is to find a winning strategy with maximum worst-case reward, where the worst-case reward is the minimum reward over all inputs. In the stochastic form of the game, an additional probabilistic player "Nature" is introduced, and the objective is to find a winning strategy with the maximum average-case reward, where the average is the expectation taken over the induced probability space. Precise definitions of these concepts can be found in [6].

Worst-case optimality We return to the example discussed in the Introduction. There, we had assumed for simplicity that each action set is assigned a cost that is its cardinality. However, the reasoning carries over to any cost function that is monotonic with respect to set inclusion: i.e., if $A \subset B$ then $\mathsf{cost}(A) < \mathsf{cost}(B)$. Intuitively, monotonicity captures the preference for choosing a smaller set of output actions. Consider the mean-payoff cost of an infinite execution where the input a is always true. For the non-compact program in Figure 1(i), it is obvious that the limit of the average cost is $\mathsf{cost}(\{b\})$. That is also the case for the compact program in Figure 1(ii): the fact that the initial cost is $\mathsf{cost}(\emptyset)$ is swamped in the limit. This is the worst case input for both programs by the monotonicity of the cost function. The best case for the program on the right is when the input a is almost everywhere false. Thus, worst-case optimality cannot distinguish between the two programs for *any* monotonic cost function.

Average-case optimality We now show that average-case optimality also cannot always distinguish between compact and non-compact strategies. The general principle is that if a strategy is non-compact only for a finite prefix of a computation, its average-case cost in the limit will be the same as the cost of a strategy which performs in a compact manner throughout.

Consider the input set $I = \{0, 1\}$. Suppose that inputs are chosen uniformly at random. The output set O is the set of subsets of the action set $A = \{a, b\}$. Let the specification be the following: the initial choice of output set is either $\{a\}$ or A; all subsequent outputs must be A. There are only two winning strategies, which differ only in their choice of initial output (either $\{a\}$ or A); both produce output A subsequently regardless of the input. Assuming unit cost per output action, the average cost of a run of length n is thus $(1+2(n-1))/n$ for the first strategy and 2 for the second. In the limit, both strategies have average cost 2, although the first is compact, while the second is not. This argument also applies for an arbitrary but monotone cost function.

In our view, quantitative measures are best suited to modeling the real cost of actions rather than to modeling a preference ordering. The two may, however, be combined to good effect. As compactness is ensured with a specification transformation, one can modularly apply quantitative synthesis to the transformed specification $\min(L, \prec)$ to obtain strategies that are compact and also optimal with respect to a cost metric.

3.3 Approximating Compactness

The worst-case exponential blowups can make it difficult to produce compact strategies. Moreover, Theorem 1 asserts that there are specifications that are realizable but have no compact strategies. For both reasons, we describe methods by which one can approximate the compactness criterion.

Approximately Minimal Languages The first method is to tighten the language L to L' that lies between L and $\min(L, \prec)$; we call L' *approximately minimal* for L. We synthesize a program satisfying L'. Given an NBA $\mathcal{A}$ for L over alphabet $I \times O$, we construct an NBA $\hat{\mathcal{A}}$ whose language is approximately minimal for L. This construction applies only to a class of preference relations that are induced pointwise by a partial order $\leq$ on individual letters of the output set O.

For infinite i/o words $w = (a, b)$ and $w' = (a', b')$, define $w \prec_p w'$ iff (1) for all i, $a_i = a'_i$ (inputs are identical) and $b_i \leq b'_i$, and (2) there is some i for which $b_i < b'_i$. We say that $w \preceq_p w'$ if $w \prec_p w'$ or $w = w'$. The ordering $\prec_p$ is transitive and regular. It is easy to construct an automaton accepting $\prec_p$, which checks condition (1) at each position of the zipped word $w \bowtie w'$, and accepts only if condition (2) holds at some position on the zipped word. The subset and cardinality preference relations introduced earlier are of this type.

Given an NBA $\mathcal{A}$ recognizing L, the NBA $\hat{\mathcal{A}}$ is constructed by excluding certain transitions of $\mathcal{A}$. Specifically, a transition $(q, (a_1, b_1), q')$ of $\mathcal{A}$ is omitted in $\hat{\mathcal{A}}$ if there is a "better" transition $(q, (a_2, b_2), q')$ in $\mathcal{A}$ with $a_1 = a_2$ and $b_2 < b_1$. Automaton $\hat{\mathcal{A}}$ can be efficiently constructed from $\mathcal{A}$ by performing a single pass over δ. The set of states, initial states and final states are identical in $\mathcal{A}$ and $\hat{\mathcal{A}}$.

Theorem 3. *For a pointwise preference order $\leq$ over O, $\mathcal{L}(\hat{\mathcal{A}})$ is an approximately minimal language for L.*

Proof. It is easy to see that $\mathcal{L}(\hat{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$, as an accepting run in $\hat{\mathcal{A}}$ is also an accepting run in $\mathcal{A}$.

For the other inclusion, let w be in $\min(L, \prec_p)$. Then, w is also in L. Thus, there is an accepting run ρ for w in $\mathcal{A}$. If all transitions in ρ are present in $\hat{\mathcal{A}}$, then w is also in $\mathcal{L}(\hat{\mathcal{A}})$. If not, there is a transition $(q, (a_1, b_1), q')$ at the k-th step of ρ (for some k) that is not present in $\hat{\mathcal{A}}$. By construction, there must be a transition $(q, (a_1, b_2), q')$ in $\mathcal{A}$ such that $b_2 < b_1$. Now consider the run ρ' that is generated by swapping transition $(q, (a_1, b_1), q')$ with $(q, (a_1, b_2), q')$ at the k-th step. This is also an accepting run, on a word w' that is identical to w except that it has (a_1, b_2) rather than (a_1, b_1) as its k-th entry. As w' is in L and $w' \prec_p w$, it cannot be the case that w is in $\min(L, \prec_p)$, a contradiction. □

Minimal Strategies for *L* The second method searches greedily for compact strategies in a game graph for L. For strategies σ and σ', say that $\sigma \sqsubseteq \sigma'$ (read as "σ is better than σ'") if for all input sequences a, $\sigma(a) = \sigma'(a)$ or $\sigma(a) \prec^{+} \sigma'(a)$. I.e., the output on input a is using σ is at least as good as that using σ'. The minimal elements according to this ordering are called *minimal strategies* for L. It is easy to show that every compact strategy for L is a minimal strategy for L. The converse does not hold.

The greedy construction applies to a game graph for L where strategies are memoryless (e.g., if synthesis for L is a safety game or a parity game), and if the preference order is pointwise, as defined above. The core idea is simple: compute the set of winning positions; then nondeterministically and greedily extract a strategy by choosing only those transitions between successive winning positions that are output-minimal with respect to $<$. In the full version, it is shown that any strategy extracted in this manner is minimal for L.

4 Evaluation

4.1 Multi-Robot Coordination

Our original motivation to investigate compactness comes from an application to multi-robot orchestration. Due to space limitations, we describe this setting in brief. One has available multiple, heterogeneous robots, each capable of carrying out certain actions, some of which cannot be allowed to overlap. The goal is to perform specified tasks by (a) assigning robots to carry out actions and (b) sequence the actions appropriately. Tasks are described in a simple declarative language, called Resh [12], that has been implemented and used to control groups of mobile robots. A useful subset of Resh is given by the following grammar, where A is the set of action names and R is a set of robot names.

$$S := a \rightarrow R \mid S \Rightarrow S \mid S \,\&\, S \mid S | S \mid S + S$$

The interpretation of these operators is in terms of a *finite-word* input-output sequence. A term $a \rightarrow R$ is interpreted as "perform action a using one of the robots in R." For this, a control strategy chooses a robot r in R, and produces a "(begin a on r)" *output* event. Action duration is not fixed: E.g., the time taken to perform a "move to position p" action may vary as the robot maneuvers around humans. The completion signal is a "(end a on r)" event that is an *input* to the control strategy. The other operators are interpreted as $\Rightarrow$ (sequencing), & (concurrent), | (choice), and + (concurrent with both tasks starting together). The interpretation of each operator produces a regular language.

The finite-word semantics is appropriate for robotics tasks that must be performed to completion. The same observation motivates the use of $\mathsf{LTL}f$ (a finite-word variant of LTL) in [38] to specify robotics tasks. A winning control strategy is one that satisfies the semantics of the operators.

As action completions are uncontrolled, even a simple specification such as $a \rightarrow R$ is unrealizable if the completion signal is never issued by an adversarial environment. It is thus necessary to restrict the environment so that every initiated action is eventually completed. This assumption *must* be interpreted over infinite words. It has the shape of a conjunction of LTL formulas $\mathsf{G}(\text{begin}(a, r) \Rightarrow \mathsf{X}\,\mathsf{F}\,\text{end}(a, r))$ over all actions a and robots r. This can be represented by a DBA which tracks the set of pending (i.e., begun but not ended) actions. This DBA is worst-case exponential in the number of action-robot pairs, but in practice is limited by the concurrency in the specification.

In order to match the infinite-word environment constraint, the Resh system specification must be extended to infinite words. This is done by saying that an infinite word w satisfies the specification if there is a prefix x of w such that x satisfies the specification. Being a regular language, a Resh specification is representable as a DFA; this is extended to infinite words as a DBA by replacing the outgoing transitions of each final state with a self-loop on all inputs.

We have arrived at the final form of the synthesis question, which has the shape $\mathcal{E} \Rightarrow \mathcal{S}$, where $\mathcal{E}$ (the environment assumptions) and $\mathcal{S}$ (the system specification) are both representable as deterministic Büchi automata. That is precisely the general form of a GR(1) specification [31]; therefore, algorithms for GR(1) synthesis can be applied to synthesize finite-state controllers.

Implementation and Experiments. Our initial experiments in synthesis with $(\mathcal{E} \Rightarrow \mathcal{S})$ occasionally produced non-compact strategies, which motivated this exploration of compactness. We now use the modified specification $\mathcal{E} \Rightarrow \hat{\mathcal{S}}$, where the system portion is made approximately compact through the construction in Section 3.3, which preserves the GR(1) format. This specification produces compact strategies for all cases we have examined.

Our implementation of GR(1) synthesis uses a SAT solver, similar to the method of [10]; we found this to be significantly faster than BDD-based methods. As there is not a well-defined set of benchmarks for robotics or Resh specifications, we generate 500 specifications at random, producing specifications with parse-tree depth 4, biased slightly to prefer the sequencing operation (i.e., $\Rightarrow$) over the others, as is likely to be the case in practice.

The system specification is set up to have two robots. Actions are allowed to overlap, which implies that all specifications are realizable. Of the 500 specifications, the GR(1) game graph was generated for 428 (85%) within a timeout limit of 5 minutes for each specification. (The Resh-to-automaton construction uses BDDs to symbolically represent output event sets, which sometimes blows up.) All 428 game graphs are solved by the SAT-based GR(1) procedure within a timeout of 5 minutes per game. The median solution time is 3 seconds; 90% are solved within 30 seconds; and all are solved within 225 seconds. We also experimented with a small hand-designed group of specifications where certain action overlaps are forbidden, which are also resolved efficiently.

4.2 Compactness for LTL

We now describe an implementation of a compact synthesis pipeline for general LTL specifications. Our experiments use the benchmarks from the SYNTCOMP (2020) competition.[4] In these experiments, the preference order is fixed as the pointwise subset order. We were forced to make this arbitrary choice as there is limited information about the origin of the benchmark problems, so we could not tailor the ordering to the problem domain.

The goal is (1) to determine the difficulty of constructing a compact synthesis pipeline for LTL, and (2) to gauge the practical feasibility of the compact synthesis procedures. The experiments are designed to answer the following questions that arise from (2): **(Q1)** What is the overhead on generating compact strategies compared to standard synthesis? **(Q2)** Is the approximation procedure more efficient than exact compactness? and **(Q3)** How effective are the approximate constructions at producing compact strategies?

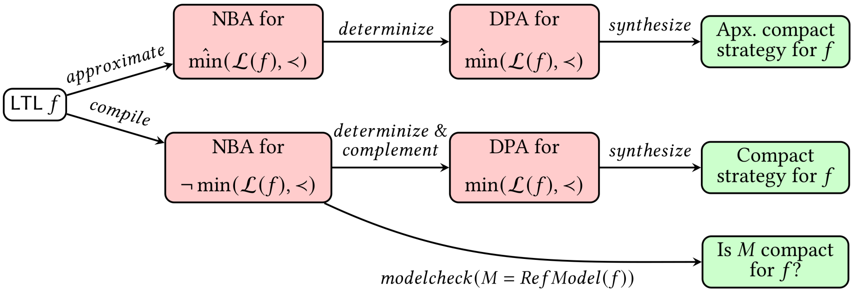

LTL f
approximate
compile
NBA for $\hat{\min}(\mathcal{L}(f), \prec)$
determinize
DPA for $\hat{\min}(\mathcal{L}(f), \prec)$
synthesize
Apx. compact strategy for f
NBA for $\neg\min(\mathcal{L}(f), \prec)$
determinize & complement
DPA for $\min(\mathcal{L}(f), \prec)$
synthesize
Compact strategy for f
$modelcheck(M = RefModel(f))$
Is M compact for f?

Fig. 2. An overview of the workflow for our experiments and tool. In the figure, $\hat{\min}$ refers to the approximate minimal language, while $RefModel(f)$ refers to the reference model for formula f.

A high-level overview of the internal structure of our tool is in Fig. 2. Our implementation chains together several known tools: the automaton libraries SPOT (v. 2.9.5) and Owl (v.20.06)[25], the synthesis tool Strix (v. 20.10)[30] and the model checker NuSMV (v. 2.6.0) [14]. We also use the AIGER toolkit [4] as well as the Syfco synthesis format converter[5]. We are grateful to the authors for making these tools freely available.

[4] At https://github.com/SYNTCOMP/benchmarks.

[5] At https://github.com/reactive-systems/syfco

Our tool offers three main features: (1) **Compact Realizability:** given an LTL formula f, determine if f is compactly realizable. This feature uses the compactness transformation from Section 3.1 to produce an automaton for the complement of the minimal language, which is then complemented, determinized and synthesized using Strix; (2) **Compactness Test:** given an LTL formula f and a candidate program P, determine if P is a compact program for f; and (3) **Approximate Compact Realizability:** given an LTL formula f, generate an approximately compact strategy. Here we implement the construction of the approximate minimal automaton from Section 3.3.

Our experiments were carried out on a Linux VM running Ubuntu 20.04 with 12 GB of memory. Naturally, we only consider synthesizing compact strategies for specifications that are realizable [6]. The results can be summarized as follows: **(A1)** We compare the efficiency of compact synthesis to the standard synthesis by evaluating the number of specifications that can be synthesized within a certain time limit. We fix this time limit to be 10 minutes, and use Strix for standard synthesis. (With this limit, the entire run over the benchmarks takes several hours.) Strix determines realizability for 396 specifications out of 421 ($\sim 94\%$), while our tool determines *compact* realizability for 213 ($\sim 50\%$). **(A2)** Within the same time limit, the approximation technique determines realizability for 398 specifications, significantly more than for exact compactness and about the same as for standard realizability. **(A3)** We model-check the strategies generated through approximate compact realizability. Model-checking for compactness requires the complement minimal automaton of a specification, so we set the time limit of 10 minutes per specification to generate this automaton. Within this limit, our tool manages to construct the required automaton for 246 specifications. Generating approximate compact strategies for these 246 specifications, and applying the Compactness Test on these strategies, we find that $\sim 42\%$ of the synthesized strategies are compact.

In addition, we tried our tool on the generalized version of the example specification from the introduction ($\bigwedge i : \mathsf{G}(a(i) \Rightarrow \mathsf{X}\, b(i))$). Our tool can synthesize a compact strategy till $N = 8$ fairly quickly, after which our setup struggles to compile the original LTL formula to an NBA. On the same set of specifications, the approximate techniques also produce a compact strategy.

The implementation process was fairly straightforward, a pleasant surprise given the number of tools and format conversions involved. We had to patch some tools to extend their capabilities (e.g., to allow automata as specifications) and to implement format conversions.

In summary, compact synthesis is feasible for a substantial number of specifications. Where it is not – due either to blowups in automaton construction or due to the gap between normal and compact realizability – one can use the approximation procedure defined in Section 3.3 to generate strategies that are minimal with respect to the strategy ordering.

[6] We refer to the helpful classification of these benchmarks into realizable and unrealizable ones from https://github.com/meyerphi/syntcomp-reference/.

5 Related Work

We discuss closely related work in synthesis and commonsense reasoning.

Qualitative Temporal Synthesis There is a considerable literature on the synthesis of open reactive programs from LTL specifications, starting with the seminal work by Pnueli and Rosner [32]. The beautiful theoretical results are made practical by the discovery of efficient algorithms for the GR(1) subclass [9,31], and procedures for bounded synthesis [21,36], based on so-called "Safraless" procedures [26]. These algorithms have been implemented in several tools, e.g., [11,15,16,19,23,33,27]. Our work builds on this basis by transforming the search for compact strategies to a standard synthesis question that can be handled by these tools.

In the robotics domain, prior work investigates synthesis for an interpretation of LTL over finite words called LTLf [22,38,40]. Although Resh is similarly restricted to finite-word properties, a central difference is that specifications in LTLf (like LTL) are defined over propositions on robot and world state, and not in terms of actions of an unknown duration.

There are many ways to choose between satisfying models: e.g., [7] designs synthesis procedures that produce minimally vacuous models. While the formulations differ, there is a common thread in the notion of minimality with respect to an ordering over models.

Quantitative Temporal Synthesis A substantial body of work in temporal synthesis is focused on *quantitative* objectives. These problems are represented by games where each action has an associated cost (or, dually, reward) and the objective is to find strategies that minimize cost (or, maximize reward) (cf. [6]). There are several ways to formulate appropriate cost/reward functions and correspondingly many ways to solve such games. One could attempt to model compactness by assigning costs to actions such that if word x is better than word y then x has the lower cost. We chose not to develop solutions along such quantitative lines for two main reasons: first, as the connection between cost and preference is indirect, setting up the right cost assignments to model a desired preference ordering is difficult; secondly, the theoretical complexity and practical difficulty of quantitative synthesis is high. Instead, we chose to tackle the question in a qualitative manner.

As shown in Section 3, quantitative measures cannot always differentiate between compact and non-compact solutions. Using the specification transformation developed here, the two methods can, however, be used in cooperation: one can model the real costs of actions in a manner that is orthogonal to the preference ordering and compute minimal-cost, compact strategies.

A recent work [1] focuses on the "quality" of satisfaction of an LTL formula (e.g., preferring to satisfy one part of a specification over another). Synthesis is through a reduction to a standard LTL specification; unfortunately this has a worst-case exponential blowup.

Non-Monotonic Reasoning. As mentioned briefly in the introduction, the compactness criterion is a form of commonsense reasoning: one does not expect synthesized solutions to include unnecessary actions. Commonsense reasoning is exemplified by the classical *frame problem*, introduced in [29], which shows that the freedom of interpretation given by logic must be restricted in order to achieve commonsense conclusions.

It was soon recognized that such restrictions imply a non-standard notion of deduction, which is not monotonic: adding new hypotheses can invalidate current conclusions [37]. In [28], McCarthy suggests a formulation in terms of a *circumscription* operation: each inference is guarded with a "not(abnormal)" predicate, and a successor state is one where the extent of this predicate is minimized—i.e., abnormal effects are maximally limited while avoiding inconsistencies. Logically, this is specified in second-order logic as $\varphi(A) \wedge \neg(\exists B : B \subset A \wedge \varphi(B))$, where φ is the specification and A is the abnormality predicate. Readers will immediately notice the similarity to the definition of $\min(L, \prec)$.

The importance of a general preference order in place of the fixed subset relation is laid out in [24]; the authors propose reasonable properties that any non-monotonic inference relation should meet, and show that a definition in terms of a preference ordering satisfies those properties. Our formulation of compactness is based on similar notions of minimality over a preference ordering on words. This is at the root of the non-monotonicity of compactness. These similarities hint at deeper connections between compactness and non-monotonic commonsense reasoning; we aim to investigate those in future work.

Acknowledgments. We would like to thank colleagues at Bell Labs and at NYU for interesting discussions and helpful comments regarding this work. This work was supported in part by the National Science Foundation grant CCF-1563393 and by DARPA under contract HR001120C0159. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense, the U.S. Government, or the National Science Foundation.

References

1. Almagor, S., Boker, U., Kupferman, O.: Formally reasoning about quality. J. ACM **63**(3), 24:1–24:56 (2016), https://doi.org/10.1145/2875421
2. Bansal, S., Chaudhuri, S., Vardi, M.Y.: Comparator automata in quantitative verification. In: FOSSACS. Lecture Notes in Computer Science, vol. 10803, pp. 420–437. Springer (2018)
3. Bansal, S., Namjoshi, K.S., Sa'ar, Y.: Synthesis of coordination programs from linear temporal specifications. Proc. ACM Program. Lang. **4**(POPL), 54:1–54:27 (2020)
4. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
5. Birget, J.: Partial orders on words, minimal elements of regular languages and state complexity. Theor. Comput. Sci. **119**(2), 267–291 (1993)

6. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 140–156. Springer (2009)
7. Bloem, R., Chockler, H., Ebrahimi, M., Strichman, O.: Synthesizing non-vacuous systems. In: VMCAI. Lecture Notes in Computer Science, vol. 10145, pp. 55–72. Springer (2017)
8. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY - A new requirements analysis tool with synthesis. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 425–429. Springer (2010)
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012)
10. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. In: McMillan, K.L., Rival, X. (eds.) VMCAI. LNCS, vol. 8318, pp. 1–20. Springer (2014)
11. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: Proc. of CAV. pp. 652–657 (2012)
12. Carroll, M., Namjoshi, K.S., Segall, I.: The Resh programming language for multirobot orchestration. In: 2021 IEEE International Conference on Robotics and Automation, ICRA. IEEE (2021), at https://arxiv.org/abs/2103.13921
13. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: QUASY: quantitative synthesis tool. In: TACAS. LNCS, vol. 6605, pp. 267–271. Springer (2011)
14. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model verifier. In: CAV. LNCS, vol. 1633, pp. 495–499. Springer (1999), https://nusmv.fbk.eu/
15. Ehlers, R.: Symbolic bounded synthesis. In: Proc. of CAV. pp. 365–379 (2010)
16. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Proc. of TACAS. pp. 272–275 (2011)
17. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: CAV. Lecture Notes in Computer Science, vol. 9780, pp. 333–339. Springer (2016), https://github.com/VerifiableRobotics/slugs
18. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. Elsevier and MIT Press (1990). https://doi.org/10.1016/b978-0-444-88074-1.50021-4
19. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An experimentation framework for bounded synthesis. In: Proc. of CAV. pp. 325–332 (2017)
20. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: Proc. of CAV. pp. 263–277 (2009)
21. Filiot, E., Jin, N., Raskin, J.: Compositional algorithms for LTL synthesis. In: Proc. of ATVA. pp. 112–127 (2010)
22. Giacomo, G.D., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. pp. 854–860. IJCAI/AAAI (2013)
23. Jobstmann, B., Roderick: Optimizations for LTL synthesis. In: Proc. of FMCAD. pp. 117–124 (2006)
24. Kraus, S., Lehmann, D., Magidor, M.: Nonmonotonic reasoning, preferential models and cumulative logics. Artif. Intell. **44**(1-2), 167–207 (1990)
25. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: ATVA. LNCS, vol. 11138, pp. 543–550. Springer (2018), https://owl.model.in.tum.de/

26. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. of FOCS. pp. 531–540. IEEE, IEEE (2005)
27. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. Softw. Syst. Model. **20**(5), 1553–1586 (2021). https://doi.org/10.1007/s10270-021-00868-z
28. McCarthy, J.: Circumscription - A form of non-monotonic reasoning. Artif. Intell. **13**(1-2), 27–39 (1980)
29. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969)
30. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018), https://strix.model.in.tum.de
31. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. In: International Conference on VMCAI. vol. 3855, pp. 364–380. Springer, Springer (2006)
32. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Prof. of POPL. pp. 179–190 (1989)
33. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A framework for developing verification algorithms. In: Proc. of CAV. pp. 171–174 (2010)
34. Rabin, M.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Math. Soc. (141), 1–35 (1969)
35. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: ATVA. Lecture Notes in Computer Science, vol. 4762, pp. 474–488. Springer (2007)
36. Schewe, S., Finkbeiner, B.: Bounded synthesis. Proc. of ATVA pp. 474–488 (2007)
37. Strasser, C., Antonelli, G.A.: Non-monotonic Logic. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, summer 2019 edn. (2019)
38. Tabajara, L.M., Vardi, M.Y.: Partitioning techniques in LTLf synthesis. In: IJCAI. pp. 5599–5606. ijcai.org (2019)
39. Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 133–191. Elsevier and MIT Press (1990), https://doi.org/10.1016/b978-0-444-88074-1.50009-3
40. Zhu, S., Giacomo, G.D., Pu, G., Vardi, M.Y.: LTLf synthesis with fairness and stability assumptions. In: AAAI. pp. 3088–3095. AAAI Press (2020)

ZDD Boolean Synthesis*

Yi Lin ✉[1], Lucas M. Tabajara[1]**, and Moshe Y. Vardi[1]

Rice University, Houston TX 77005, USA
vardi@cs.rice.edu, l.martinelli.tabajara@gmail.com, yl182@rice.edu

Abstract. Motivated by applications in boolean-circuit design, boolean synthesis is the process of synthesizing a boolean function with multiple outputs, given a relation between its inputs and outputs. Previous work has attempted to solve boolean functional synthesis by converting a specification formula into a Binary Decision Diagram (BDD) and quantifying existentially the output variables. We make use of the fact that the specification is usually given in the form of a Conjunctive Normal Form (CNF) formula, and we can perform resolution on a symbolic representation of a CNF formula in the form of a Zero-suppressed Binary Decision Diagram (ZDD). We adapt the realizability test to the context of CNF and ZDD, and show that the *Cross* operation defined in earlier work can be used for witness construction. Experiments show that our approach is complementary to BDD-based Boolean synthesis.

Keywords: Boolean synthesis · Binary decision diagram · Zero-suppressed binary decision diagram · Quantifier elimination · Resolution.

1 Introduction

Boolean functions are widely used in electronic circuits, and thus in many aspects of computing, to describe operations over binary values. Often the most natural way to express such an operation is as a declarative relation between inputs and outputs. Implementing these operations in practice, however, requires a functional, rather than declarative, representation. The process of constructing a function that generates outputs directly from inputs, based on a given declarative relation between them, is called *boolean synthesis*. For example, boolean synthesis can be applied in constructing a full logical circuit from a relational specification [9,15] or an unknown intermediate component in an existing logical circuit [12]. Boolean synthesis is also useful for computing certificates for quantified boolean formulas (QBF), and advances in QBF solving and boolean synthesis are motivated by each other [3,20].

Formally, we are given a specification $f(\vec{x}, \vec{y})$, from $\mathbb{B}^m \times \mathbb{B}^n$ to $\mathbb{B}$, relating two sets of boolean variables. The specification holds true if and only if $\vec{y}$ is a

* Work supported in part by NSF grants CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office.

** Currently at Runtime Verification, Inc.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 64–83, 2022.
https://doi.org/10.1007/978-3-030-99524-9_4

correct output for the inputs $\vec{x}$. We solve the synthesis problem following the convention of splitting it into two sub-problems [9]:

1. *Realizability*: constructing the realizable set $R \subseteq \mathbb{B}^m$ of input assignments $\vec{x}$ for which there exists an output assignment $\vec{y}$ such that $f(\vec{x}, \vec{y}) = 1$.
2. *Witness construction*: constructing a witness function $g : \mathbb{B}^m \to \mathbb{B}^n$ that computes an output $\vec{y} = g(\vec{x})$ from an input $\vec{x} \in R$ such that $f(\vec{x}, \vec{y}) = 1$.

Given a propositional formula f as the relational specification, we aim to synthesize a boolean function g that is correct by construction, meaning that as long as the input is realizable the output will satisfy the specification.

Prior work solved the boolean functional synthesis by converting the specification formula into a Binary Decision Diagram (BDD), defined in Section 2, and quantifying the output variables existentially [9]. BDDs constitute a formalism for representing Boolean functions, supported by mature tools such as CUDD [22]. The size of a BDD representing a formula can, however, be exponential in the number of variables. Oftentimes, it is even not possible to construct the BDD before starting to solve the problem [9]. Noticing how this blow-up in BDD size has restricted the potential of existing BDD-based synthesis algorithms, we seek to develop an algorithm that reduces the impact of this exponential blowup. Hence we look for an alternative data structure that might be more promising in representing boolean formulas compactly.

We identify here Zero-Suppressed Binary Decision Diagram (ZDD) [16], defined in Section 2, as such an alternative approach. ZDDs have been shown to sometimes outperform BDDs in the context of QBF solving [19]. Unlike BDDs, which represent a boolean formula *semantically* via the set of satisfying assignments, ZDDs are designed to encode sets of sets [14], allowing them to represent *syntactically* a formula in Conjunctive Normal Form (CNF) as a set of clauses, which are themselves sets of literals. This means that it may require an exponential-size BDD to represent a CNF formula, which can be alternatively compactly encoded as a polynomial-size ZDD representation.

It can be expected, however, that this more compact representation comes at a cost. Since ZDDs do not represent the solution sets directly like BDDs do, solving realizability and synthesis over this representation might require additional effort. With this in mind. we perform here a full investigation comparing ZDDs and BDDs for boolean synthesis. We focus on the following research questions:

1. How do the sizes of the ZDD and BDD representations compare, and how does this affect the time of compiling the formulas into the diagram representation? Are ZDDs always more compact?
2. In realizability, how do ZDDs vs. BDDs perform, in time and space?
3. How do ZDDs perform, compared to BDDs, in witness construction?
4. How does the end-to-end synthesis performance of ZDDs compare to BDDs?
5. For scalable families of formulas, how does the time and space performance scale as the formula grows, comparing ZDDs to BDDs?

Our synthesis problem can often be expressed as boolean synthesis for CNF specifications, as the boolean specification in synthesis problems is often given in CNF form, and even non-CNF specifications can be easily converted to CNF. Once specification formulas are given in CNF, it is possible to perform realizability by using the *resolution* operation, which is equivalent to existentially quantifying the output variables directly from the CNF formula. Each resolution step increases the number of clauses quadratically. But when a ZDD is used to represent the CNF formula, even when the number of clauses increases quadratically, the size of the ZDD tends to increase to a lesser extent.

The crux of our contribution is a boolean-synthesis algorithm that performs resolution on a symbolic, ZDD-based representation of CNF formulas. To solve the first sub-problem of realizability, we compute the set $R \subseteq \mathbb{B}^m$ of all realizable inputs, and then check the full and partial realizability of the input domain. The realizable set is generated by applying resolution to the ZDD representation of the CNF formula, based on operations defined in previous work [4,5,19].

The second sub-problem requires construction of a witness function $g : \mathbb{B}^m \rightarrow \mathbb{B}^n$ for the output variables $\vec{y} \in \mathbb{B}^n$. We adapt the formulas defined in previous work [9] to the context of CNF, eliminating one output variable $y_i \in \mathbb{B}$ at a time, and make use of the fact that resolution is equivalent to existential quantification. In this way we can extract a witness $g_i : \mathbb{B}^m \rightarrow \mathbb{B}$ for variable y_i without abandoning the ZDD representation.

After substituting the witness of an output variable back in the formula, we need to compute the next witness. This leads to our next challenge, which is how to guarantee that the formula remains in CNF after performing this substitution. The overall form of the entire formula after substitution is dependent on the form of the substituted witness function g_i: clauses where y_i is positive can be converted back to CNF if g_i is also in CNF, but clauses where y_i is negative require g_i to be in *Disjunctive* Normal Form (DNF). Thus, what we need are two equivalent witness functions, one in CNF and the other in DNF.

Our solution is to use the *Cross* ZDD operation, first defined by Knuth [14]. We show that if the *Cross* operation, defined on "families of sets" [14], is applied to a ZDD representation of a CNF formula, then the result can be interpreted as the ZDD for an equivalent DNF. In this way, with the *Cross* operation, we can use the CNF version of a witness for positive occurrences of a variable, and use the equivalent DNF version for negative occurrences, while both preserving the equivalence and ensuring that the resulting formula remains in CNF.

Our experimental evaluation confirms the advantages of ZDDs in compilation, thanks to their linear size and direct correspondence to the CNF formula structure. As expected, this more compact representation can come with a tradeoff of increasing the difficulty of constructing witnesses. Therefore, in synthesis performance, neither ZDDs nor BDDs dominate across the board, each performing better in different families of formulas. We therefore advocate for the ZDD-based approach as an addition to the portfolio of boolean synthesis tools, serving as a complement to BDD-based approaches [11].

As shown in related works on boolean synthesis, there exist alternative tools including CegarSkolem [13], BFSS [1] and Manthan [10]. Our focus of comparison here is, however, on improvements to decision-diagrams based tools for boolean synthesis, rather than tools based, for example, on QBF solvers. Decision-diagram based approaches enjoy some unique advantage. For example, decision diagrams facilitate partitioned-form representation [23]. Also, decision diagrams can be used as intermediate-step representation in temporal synthesis [24]. These unique advantages justify, we believe, our focus here on decision-diagram based approaches. We return to this point in our discussion of future work.

2 Preliminaries

Boolean Formulas and Functions. Boolean formulas and boolean functions are built upon the boolean set $\mathbb{B} = \{0, 1\}$. We identify a boolean formula $f(\vec{x})$ over m propositional variables $\vec{x} = (x_1, \dots, x_m)$ with the boolean function $f : \mathbb{B}^m \to \mathbb{B}$ such that $f(\vec{a}) = 1$ for an assignment $\vec{a} = (a_1, \dots, a_m) \in \mathbb{B}^m$ if and only if $\vec{a}$ is a satisfying assignment to $\vec{x}$ in the formula. Two boolean formulas f and f' are logically equivalent if they represent the same boolean function (and therefore have the same set of satisfying assignments). Substitution of a boolean expression $d(\vec{x})$ in place of a variable x_i in a boolean formula $f(\vec{x})$ is denoted by $f[x_i \mapsto d]$ and defined by $f[x_i \mapsto d](\vec{x}) = f(x_1, \dots, x_{i-1}, d(\vec{x}), x_{i+1}, \dots, x_m)$.

Conjunctive and Disjunctive Normal Forms. A *literal* is either a variable or the negation of a variable. A *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. A boolean formula in the form of a conjunction of clauses is said to be in *Conjunctive Normal Form* (CNF), and a boolean formula in the form of a disjunction of cubes is said to be in *Disjunctive Normal Form* (DNF).

Definition 1 (Boolean Synthesis Problem). *Given a boolean formula $f(\vec{x}, \vec{y})$ in CNF with $m + n$ boolean variables, partitioned into m* input variables *$\vec{x} = (x_1, \dots, x_m)$ and n* output variables *$\vec{y} = (y_1, \dots, y_n)$, construct:*

1. *The set $R \subseteq \mathbb{B}^m$, called the* realizability set*, of all assignments $\vec{a} \in \mathbb{B}^m$ to $\vec{x}$ for which there exists an assignment $\vec{b} \in \mathbb{B}^n$ to $\vec{y}$ such that $f(\vec{a}, \vec{b}) = 1$.*
2. *A function $g : \mathbb{B}^m \to \mathbb{B}^n$ such that $f(\vec{a}, g(\vec{a})) = 1$ for all $\vec{a} \in R$. This is called a* witness function*. In practice, arbitrary formulas can be converted to equi-realizable CNF formulas with a linear blowup using Tseytin encoding, quantifying existentially over Tseytin variables. The witnesses for the equi-realizable formula can then be used for the original formula.*

Binary Decision Diagrams. A (Reduced Ordered) Binary Decision Diagram (BDD) [2] is a directed acyclic graph that represents a boolean function. Internal nodes of the BDD represent boolean variables, and paths on the BDD correspond to assignments, leading either to a terminal node 1 if satisfying or 0 if unsatisfying. We assume that all BDDs are *ordered*, meaning that variables are ordered in the same way along every path, and *reduced*, meaning that superfluous nodes are removed and identical subgraphs are merged. Given these two conditions, BDDs are a *canonical* representation, meaning that two BDDs with the same variable order that represent the same function will be identical [2]. The variable order used can have a major impact on the BDD's size, and two BDDs representing the same function but with different orders can have an exponential difference in size.

Zero-Suppressed Decision Diagrams. A Zero-Suppressed Binary Decision Diagram (ZDD), is a data structure first defined in [16]. ZDDs are similar to BDDs but use a different reduction rule: while BDDs remove nodes where both edges point to the same child, ZDDs remove nodes where the 1-edge (edge assigning the variable to 1) points directly to the 0-terminal. Specifically, the 0-ZDD encodes formulas that are always valid, and the 1-ZDD encodes contradiction.

Semantics on Families of Sets. ZDDs can be used to implicitly represent families of subsets of a set S, where the variables in the ZDD correspond to elements of S that can be either present or absent in a subset [14]. For a ZDD Z, we denote by $[\![Z]\!]$ the family of subsets represented by Z. We define $[\![0]\!] = \varnothing$ and $[\![1]\!] = \{\varnothing\}$ for the terminals 0 and 1, respectively. Using $Z(x, Z_0, Z_1)$ to denote a ZDD with variable x as the root, ZDD Z_0 as the 0-child and ZDD Z_1 as the 1-child, we define $[\![Z(x, Z_0, Z_1)]\!] = [\![Z_0]\!] \cup \{\{x\} \cup \alpha \mid \alpha \in [\![Z_1]\!]\}$. Note that using this interpretation every subset in the family corresponds to a path to the terminal 1 on the ZDD. Since CNF formulas can be viewed as sets of clauses, where a clause can be viewed as a set of literals, we can use ZDDs to represent CNF formulas syntactically. When representing a formula in CNF by a ZDD, for each atomic proposition p we treat its positive and negative literals p and $(\neg p)$ as two distinct variables x_p and $x_{\neg p}$. Then every path leading to the 1-terminal corresponds to a clause in the CNF formula, where x_l connects to its 1-edge in the path if and only if the literal l is in the corresponding clause.

ZDD Operations. We use standard ZDD operations such as *Subset0*, *Subset1*, *Change*, *Union*, *Intersect*, and *Difference*, defined previously in [17] and implemented in the CUDD package [22]. In terms of families of sets, $\mathit{Subset0}(Z, x)$ returns the family of all sets α such that $\alpha \in [\![Z]\!]$ and $x \notin \alpha$, and $\mathit{Subset1}(Z, x)$ returns the family of all sets $\alpha \setminus \{x\}$ such that $\alpha \in [\![Z]\!]$ and $x \in \alpha$. $\mathit{Change}(Z, x)$ returns the family $\{\alpha \cup \{x\} \mid \alpha \in [\![Z]\!] \text{ and } x \notin \alpha\} \cup \{\alpha \setminus \{x\} \mid \alpha \in [\![Z]\!] \text{ and } x \in \alpha\}$. The operation *Resolution(x, Z)* returns the ZDD representing the result of applying resolution to variable x in the CNF represented by Z. It is implemented following [4], using the operations *SubsumptionFreeUnion*, which takes the union of two families of sets while removing subsumed sets, and *ClauseDistribution*, which returns the family of sets resulting from applying distribution over two given sets of clauses. The witness-construction phase also requires the *Cross* operation defined in [14] to convert between CNF and DNF representations. See Section 4 for details.

3 Realizability Using ZDDs

We describe in this work a ZDD-based algorithm to solve the Boolean-Synthesis Problem described in Definition 1. This means that the specification $f(\vec{x}, \vec{y})$, the realizability set R and the witness function g are all represented by CNF formulas encoded as ZDDs, as defined in Section 2. In this section we describe how to compute the realizability set R and analyze it to answer whether the specification is partially or fully realizable. In Section 4 we describe how to compute the witness function g.

3.1 Realizable Set R

In order to construct the set R of realizable assignments to the input variables $\vec{x}$, as described in Definition 1, we need to quantify existentially the output variables $\vec{y}$, analogously to the BDD-based approach of [9].

Let $f_0, \ldots, f_n$ be CNF formulas such that

$$\begin{aligned}
&f_n \equiv f\\
&f_{n-1} \equiv (\exists y_n) f\\
&\ldots\\
&f_i \equiv (\exists y_{i+1}) \ldots (\exists y_{n-1})(\exists y_n) f\\
&\ldots\\
&f_1 \equiv (\exists y_2) \ldots (\exists y_{n-1})(\exists y_n) f\\
&f_0 \equiv (\exists y_1) \ldots (\exists y_{n-1})(\exists y_n) f
\end{aligned}$$

As in [9], the last formula $f_0 = (\exists y_1) \ldots (\exists y_{n-1})(\exists y_n) f$ implicitly represents the realizable set R, describing the set of satisfying assignments of f_0.

To compute $f_0, \ldots, f_n$ as CNF formulas, we apply the *resolution* operation, which is equivalent to existential quantification [7]. We first state a normal-form lemma.

Lemma 1. [4] *Let f be a CNF formula. Let f_p^+ denote the conjunction of all clauses α such that $(p \vee \alpha)$ is a clause in f. Let f_p^- denote the conjunction of all clauses β such that $((\neg p) \vee \beta)$ is a clause in f. Let f_p' denote the conjunction of clauses γ in f where neither p nor $(\neg p)$ is a literal in γ. Then f is logically equivalent to $(p \vee f_p^+) \wedge (\neg p \vee f_p^-) \wedge f_p'$ for a boolean variable p.*

Proof. The claim follows from [4]. □

Next we show how to use resolution to existentially quantify a variable from a formula in the normal form of Lemma 1.

Lemma 2. *Let y be a boolean variable, then the boolean formula $(\exists y)((y \vee f_y^+) \wedge (\neg y \vee f_y^-) \wedge f_y')$ is logically equivalent to $((f_y^+ \vee f_y^-) \wedge f_y')$.*

Proof.

$$\begin{aligned}
&(\exists y)((y \vee f_y^+) \wedge (\neg y \vee f_y^-) \wedge f'_y) \\
&\equiv (\exists y)(((y \wedge \neg y) \vee (y \wedge f_y^-) \vee (f_y^+ \wedge \neg y) \vee (f_y^+ \wedge f_y^-)) \wedge f'_y) \\
&\equiv ((\exists y)(y \wedge f_y^-) \vee (\exists y)(f_y^+ \wedge \neg y) \vee (\exists y)(f_y^+ \wedge f_y^-)) \wedge f'_y && (f'_y \text{ excludes } y) \\
&\equiv (((\exists y)(y \wedge f_y^-) \vee (\exists y)(f_y^+ \wedge \neg y) \vee (f_y^+ \wedge f_y^-)) \wedge f'_y) && (f_y^+,\ f_y^- \text{ excludes } y) \\
&\equiv ((f_y^- \vee f_y^+ \vee (f_y^+ \wedge f_y^-)) \wedge f'_y) && (f_y^+,\ f_y^- \text{ excludes } y) \\
&\equiv ((f_y^- \vee f_y^+) \wedge f'_y)
\end{aligned}$$

□

We call the formula $((f_y^+ \vee f_y^-) \wedge f'_y)$ the *resolution of the variable* y *in* f. Note that this formula (specifically the subformula $(f_y^+ \vee f_y^-)$) is not in CNF, but can be easily rewritten in CNF by distributing the clauses in f_y^+ over the clauses in f_y^-. The equivalence of resolution and existential quantification then follows from Lemmas 1 and 2 above:

Corollary 1. *For a formula* f *and a boolean variable* y*, the formula* $(\exists y)f$ *is logically equivalent to* $((f_y^+ \vee f_y^-) \wedge f'_y)$*.*

Proof. The claim follows from Lemmas 1 and 2. □

We represent f_y^+, f_y^-, f'_y by ZDDs by applying the *Subset0* and *Subset1* operations described in Section 2: $Z_y^+ = Subset1(Z, y)$, $Z_y^- = Subset1(Z, \neg y)$, and $Z'_y = Subset0(Subset0(Z, y), \neg y)$. We then use the *ClauseDistribution* operation to distribute the clauses of Z_y^+ over Z_y^-, and the *SubsumptionFreeUnion* operation to combine all clauses into a single ZDD. This implements the operation $Resolution(y_i, Z)$ mentioned in Section 2. In practice, we follow the *Cut-Elimination Algorithm* of [4], which also eliminates tautologies by removing clauses where the same variable appears both positively and negatively. Therefore we can assume that the ZDD representations of $f_0, \ldots, f_n$ do not include subsumed and tautological clauses, which may also lead to smaller ZDDs.

The advantage of applying resolution symbolically over a ZDD representation, rather than directly over the CNF formula is that every resolution step increases the number of clauses in the formula quadratically. Thus, the number of clauses after multiple resolution steps can easily grow exponentially. ZDDs, compared to representing clauses explicitly, are well-equipped for representing compactly large sets of clauses, often being able to represent an exponential set of clauses in polynomial space [16]. The ZDD representation also makes it easy to remove subsumed and tautological clauses, further reducing size.

3.2 Full and Partial Realizability

When the realizable set R is represented by a BDD, as in [9], it is easy to check whether $R = \varnothing$ or $R = \mathbb{B}^m$, as this corresponds to the BDD being equal to 0 or 1, respectively. This is less straightforward for a ZDD representation, which expresses R indirectly by the set of clauses in its CNF representation f_0, rather than the set of assignments itself. We say that a CNF specification $f(\vec{x}, \vec{y})$ is *fully realizable* if and only if all $\vec{a} \in \mathbb{B}^m$ have some $\vec{b} \in \mathbb{B}^n$ so that $f(\vec{a}, \vec{b})$ holds. This corresponds to $R = \mathbb{B}^m$. Similarly, we say that f is *partially realizable* if and only if there is some $\vec{a}$ for which there exists some $\vec{b}$ so that $f(\vec{a}, \vec{b})$ holds. This corresponds to $R \neq \varnothing$. After computing a ZDD representation of R, we wish to check full and partial realizability over this representation.

Theorem 1. *The CNF specification $f(\vec{x}, \vec{y})$ is fully realizable if and only if the ZDD for f_0 is equivalent to the 0-ZDD.*

Proof. The specification $f(\vec{x}, \vec{y})$ is fully realizable if and only if the CNF formula f_0 representing R is a tautology, which means that every clause of R has both p and $\neg p$ for some variable p, i.e., every clause is a tautology. Tautologies, however, are automatically removed by the *Resolution* operation, as explained in Section 3.1. Thus, full realizability occurs if and only if the set of clauses is empty, represented by the ZDD 0. □

Note that the realizability R is represented by the CNF formula $f_0 \equiv (\exists y_1) \ldots (\exists y_n) f$, which does not contain any free occurrences of $\vec{y}$ variables. We then perform resolution on the $\vec{x}$ variables in the same way as we did for the $\vec{y}$ variables. Then the original formula is partially realizable if and only if $(\exists x_1)(\exists x_2) \ldots (\exists x_m) f_0$ is true, meaning that resolution does not derive a contradiction. If a contradiction is derived, the resulting ZDD is the terminal 1, representing the empty clause. Otherwise it is the terminal 0.

Theorem 2. *The CNF specification $f(\vec{x}, \vec{y})$ is partially realizable if and only if the ZDD representing $(\exists x_1)(\exists x_2) \ldots (\exists x_m) f_0$ is equivalent to the 0-ZDD.*

Proof. Since all variables are existentially quantified, the ZDD must be either the terminal 0 (representing the empty CNF, equivalent to *true*) or the terminal 1 (representing a CNF with an empty clause, equivalent to *false*). In the first case, the formula $(\exists x_1) \ldots (\exists x_m)(\exists y_1) \ldots (\exists y_n) f$ is true, meaning that there is an assignment that satisfies $f(\vec{x}, \vec{y})$, which by definition makes f partially realizable. In the second case, the formula is false, meaning that there is no such assignment, and therefore f is not partially realizable. □

4 Synthesis Using ZDDs

As described in [9], once we have computed the formulas $f_1, \ldots, f_n$ with the output variables existentially quantified, we can construct the witness g_i for variable y_i from the formula $f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}]$, after having computed the witnesses $g_1, \ldots, g_{i-1}$ for the preceding variables. In [9], two witness functions were presented for variable y_i: the default-1 witness $f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}][y_i \mapsto 1]$ and the default-0 witness $(\neg f_i)[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}][y_i \mapsto 0]$. In this work, however, we additionally want to ensure that we maintain the CNF form of the specification after substituting $g_1, \ldots, g_{i-1}$ into f_i, to enable ZDD representation.

In this section we show how to construct and substitute witnesses so that the result remains in CNF.

For ZDD-based algorithms, the iterated substitution approach requires more sophistication for the construction of the witnesses, compared to the iterated-substitution approach for BDDs. We solve this problem in Section 4.2. As in [9], the resulting witnesses guarantee that $f(\vec{a}, g_1(\vec{a}), \ldots, g_n(\vec{a})) = 1$ for all realizable input assignments $\vec{a} \in R$.

4.1 Witnesses for Single-Dimension Output Variable

As in [9], we start by defining witnesses for the case when there is a single output variable:

Lemma 3. *Let f be a CNF formula over boolean variables $x_1, \ldots, x_m, y$. Then the formulas f_y^- and $\neg f_y^+$ are witnesses for the variable y.*

Proof. The realizability set, as defined in Section 3.1, is $R = \{\vec{a} \in \mathbb{B}^m \mid (\exists y) f[\vec{x} \mapsto \vec{a}] \equiv 1\}$. Thus, by Corollary 1, for all $\vec{a} \in R$

$$((f_y^+ \vee f_y^-) \wedge f_y')[\vec{x} \mapsto \vec{a}] \equiv 1. \tag{1}$$

Hence $f_y'[\vec{x} \mapsto \vec{a}] \equiv 1$ and either $f_y^+[\vec{x} \mapsto \vec{a}] \equiv 1$ or $f_y^-[\vec{x} \mapsto \vec{a}] \equiv 1$.

Now we want to show $f(\vec{a}, g(\vec{a})) = 1$, i.e., $f[y \mapsto g(\vec{x})][\vec{x} \mapsto \vec{a}] = 1$, for both $g(\vec{x}) = f_y^-$ and $g(\vec{x}) = \neg f_y^+$.

For $g(\vec{x}) = f_y^-$, since $f \equiv f_y' \wedge (y \vee f_y^+) \wedge ((\neg y) \vee f_y^-)$, we are left to show $f[y \mapsto f_y^-][\vec{x} \mapsto \vec{a}] \equiv (f_y' \wedge (f_y^- \vee f_y^+) \wedge ((\neg f_y^-) \vee f_y^-))[\vec{x} \mapsto \vec{a}] \equiv 1$. By (1) we are only left to show $((\neg f_y^-) \vee f_y^-)[\vec{x} \mapsto \vec{a}] \equiv 1$, which follows from the left-hand side being a tautology.

Similarly, for $g(\vec{x}) = \neg f_y^+$, we need to show $f(\vec{a}, g(\vec{a})) = f[y \mapsto (\neg f_y^+)][\vec{x} \mapsto \vec{a}] \equiv 1$. This is equivalent to showing that $(f_y' \wedge ((\neg f_y^+) \vee f_y^+) \wedge (f_y^+ \vee f_y^-))[\vec{x} \mapsto \vec{a}] \equiv 1$. By (1) we are only left to show $((\neg f_y^+) \vee f_y^+)[\vec{x} \mapsto \vec{a}] \equiv 1$, a tautology.

□

Note that the witness f_y^- is in CNF, while the witness $\neg f_y^+$, being the negation of a CNF formula, can be more easily represented in DNF. Note also that these witnesses do not correspond exactly to the default-1 and default-0 witnesses of [9], which would more specifically be equivalent to $f_y^- \wedge f_y'$ and $\neg(f_y^+ \wedge f_y')$, respectively. We choose the alternative witnesses because they contain fewer clauses, and thus are more likely to produce a more efficient ZDD representation.

4.2 Preserve CNF by Equivalent Witnesses

We now explain how to construct witnesses of multiple output variables. Let $f_n, \ldots, f_0$ be as defined in Section 3.1. We can then compute a witness for each y_i iteratively, as in [9]. Using the f_y^- witness from Lemma 3, for example, this means $g_i(\vec{x}) = (f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}])_{y_i}^-$, where g_i is the witness for variable y_i.

The substitution $f_i[y \mapsto g]$, however, is not necessarily in CNF. But Lemma 3 requires that the formula is in CNF in order to extract the next witness. This means that we need to find a way to perform the substitution in a way that the result remains in CNF.

Recall that, since our *Resolution* operation removes tautological clauses, each variable can only occur in positive or negative form in a clause, but not both. If the witness g is in CNF, e.g., $g = f_y^-$, we can substitute this witness in a clause $(y \vee l_1 \vee l_2 \vee \ldots)$ where y occurs in positive form. The result is a disjunction of the literals $l_1, l_2, \ldots$ and the CNF $g = (cl_1 \wedge cl_2 \wedge \ldots)$. By distribution, we can write this as an equivalent CNF $((cl_1 \vee l_1 \vee l_2 \vee \ldots) \wedge (cl_2 \vee l_1 \vee l_2 \vee \ldots) \wedge \ldots)$. Likewise, if the witness g for y is in DNF, e.g., $g = (\neg f_y^+)$, then, after the substitution, every clause $(\neg y \vee l_1 \vee l_2 \vee \ldots)$ where y appears in negative form can be converted to the CNF $(\neg(\neg(cl_1 \wedge cl_2 \wedge \ldots)) \vee l_1 \vee l_2 \vee \ldots) \equiv ((cl_1 \wedge cl_2 \wedge \ldots) \vee l_1 \vee l_2 \vee \ldots) \equiv ((cl_1 \vee l_1 \vee l_2 \vee \ldots) \wedge (cl_2 \vee l_1 \vee l_2 \vee \ldots) \wedge \ldots)$.

The problem, therefore, is that if we want the result to be in CNF, CNF witnesses work well for positive occurrences, while DNF witnesses work well for negative occurrences. Thus, as long as we can find an efficient conversion between CNF formulas and their equivalent DNF formulas, we can ensure that the substitution formula $f_i[y \mapsto g]$ can be written as a CNF. For this purpose, we introduce the *Cross* operator from [14].

Definition 2 (*Cross* operation).
Let S be a family of sets of literals. Then

$$\mathit{Cross}(S) = \mathit{Minimal}\{t \mid \forall s_i \in S : t \cap s_i \neq \varnothing\},$$

where

$$\mathit{Minimal}(S) = \{t \in S \mid \forall s \in S : s \subseteq t \rightarrow s = t\}.$$

Hence, $\mathit{Cross}(S)$ is a family of sets of literals, such that every set t of literals in $\mathit{Cross}(S)$ has at least a common literal with every set of literals in S. Moreover, every set t in $\mathit{Cross}(S)$ is irredundant [14], meaning they are the smallest possible sets satisfying this property.

Specifically, if S represents a given CNF f, where every set $s_i \in S$ represents a clause and the elements of s_i are the literals in that clause, then $\mathit{Cross}(S)$ represents the set of smallest possible sets t such that t has at least one common literal with every disjunctive clause of f. Equivalently speaking, $\mathit{Cross}(S)$ collects all t such that every disjunctive clause is satisfied, i.e., it is a collection of all irredundant sets of literals corresponding to irredundant assignments to variables. This further means $\mathit{Cross}(S)$ is a collection of prime implicants of f [6,14], whose disjunction has been proved to be a DNF equivalent to the CNF f. Therefore, whenever a CNF is given, we can construct a set S of sets, where every set in S collects literals in a disjunctive clause of the CNF. Then $\mathit{Cross}(S)$ returns a set of sets representing an equivalent DNF. Conversely, when interpreted as a DNF, $\mathit{Cross}(S)$ is equivalent to S interpreted as a CNF.

By the analysis above, we can extend Definition 2 of the *Cross* operation to CNF formulas:

Definition 3 (CNF Cross operation). *Let f be a CNF formula $cl_1 \wedge \ldots \wedge cl_k$, where every $cl_i = \bigvee_{\ell \in L_i} \ell$ is a clause formed by the disjunction of a set of literals L_i. Let $S = \{L_1, \ldots, L_k\}$ be the representation of f as a family of sets. Then,*

$$Cross(f) = \bigvee_{L'_i \in Cross(S)} \bigwedge_{\ell \in L'_i} \ell$$

Note that $Cross(f)$ is a DNF formula. We can similarly define in an analogous way the $Cross$ of a DNF formula as a CNF formula. We can verify that $Cross(f)$ and f are equivalent:

Lemma 4. *For a CNF formula f, $Cross(f) \equiv f$.*

Proof. By analysis above, the set $Cross(S)$ includes elements $L'_i s$ which are irredundant smallest sets that each has common literal with every set of literals in S. Therefore, every conjunction $\bigwedge_{\ell \in L'_i} \ell$, or cube, has common literal with every disjunctive clauses in CNF f, and thus every cube has the same boolean values under the same set of truth assignments as a prime implicant [6,21] of CNF f. Then it follows that the DNF $Cross(f)$, as a disjunction of these conjunctions, is logically equivalent to the disjunction of all prime implicants of the CNF f, as proved by previous works [21]. □

Note that the same result also holds for DNF formulas, following from the fact that $Cross(f) \equiv f$ if and only if $\neg Cross(f) \equiv \neg f$.

Now we aim to show how to construct witnesses one by one, why this construction is correct, and why this construction is viable. First, if we fix the witness $g_j = (f_i)^-_{y_j}$, and substitute positive and negative occurrences with g_j and $Cross(g_j)$ in the CNF formula f_i, then the equivalence and CNF form of $f_i[y_j \mapsto g_j]$ can both be preserved. We use the following lemma:

Lemma 5. *Let f and g be given as CNF formulas. Then $f[y \mapsto g]$ is equivalent to $(g \vee f^+_y) \wedge (\neg Cross(g) \vee f^-_y) \wedge f'_y$.*

Proof. By Lemmas 1 and 4, $f[y \mapsto g] \equiv ((y \vee f^+_y) \wedge (\neg y \vee f^-_y) \wedge f'_y)[y \mapsto g] = (g \vee f^+_y) \wedge (\neg g \vee f^-_y) \wedge f'_y \equiv (g \vee f^+_y) \wedge (\neg Cross(g) \vee f^-_y) \wedge f'_y$.

□

Since $g = f^-_y$ is a CNF formula, $Cross(g)$ is a DNF formula, and $\neg Cross(g)$ is a CNF. By distribution of f^+_y over clauses in g, and distribution of f^-_y over clauses in $\neg Cross(g)$, the resulting expression $(g \vee f^+_y) \wedge (\neg Cross(g) \vee f^-_y) \wedge f'_y$ can be converted to CNF form.

Alternatively, we can pick the witness $g = \neg f^+_y$, and instead substitute $Cross(g)$ on positive occurrences and g on negative occurrences of y. Similarly, the formula $(Cross(g) \vee f^+_y) \wedge (\neg g \vee f^-_y) \wedge f'_y$ can also be converted to an equivalent CNF. Therefore, the equivalence and CNF form is preserved for $f_i[y_j \mapsto g_j]$, leading to the following corollary.

Corollary 2. *Every step in $g_i(\vec{x}) = (f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}])^-_{y_i}$ can be performed so it returns a CNF formula.*

Proof. Corollary 2 follows from Lemma 3, Definition 3, and Lemma 5 □

Finally, we have the witnesses constructed in this process:

Theorem 3. *Let $g_i(\vec{x}) = (f_i[y_1 \mapsto g_1]\dots[y_{i-1} \mapsto g_{i-1}])^-_{y_i}$ for $0 \leq i \leq n$. Then, g_i is a witness for y_i in f, for every y_i. The same applies if $g_i(\vec{x}) = \neg(f_i[y_1 \mapsto g_1]\dots[y_{i-1} \mapsto g_{i-1}])^+_{y_i}$*

Proof. Theorem 3 follows from Lemma 5 and Corollary 2. □

4.3 Algorithm for Constructing Witnesses

In the last subsection we described how to uses Knuth's Cross operation to facilitate CBF/DNF conversion, enabling the use of iterated substitution. We describe our novel algorithm for synthesis using ZDDs.

We start by presenting the ZDD implementation of *Cross* function from Definition 2, following [14]:

```
if ZDD Z is the 1-terminal then
    return 0-terminal;
else if ZDD Z is the 0-terminal then
    return 1-terminal;
else
    // $Z_l$ denotes the ZDD rooted at 0-child of root of $Z$
    // $Z_h$ denotes the ZDD rooted at 1-child of root of $Z$
    $Z_r = Union(Z_l, Z_h)$;
    $Z_{ll} = Cross(Z_r)$;
    $Z_r = Cross(Z_l)$;
    $Z_{hh} = Difference(Z_r, Z_{ll})$;
    // $Var(Z)$ denotes the variable at the root node of $Z$
    $Z' = NewZDD(Var(Z), Z_{ll}, Z_{hh})$;
    return Z';
end
```

We now explain how to perform the substitution following Lemma 5, where we want to construct a ZDD of $f[y \mapsto g]$, where f and g are CNF formulas and y is a variable. Denote the ZDD representation of f as Z_f and that of g as Z_g. Then we compute the ZDD $Cross(Z_g)$ using the algorithm above. Recall that this ZDD represents a DNF formula that is equivalent to g.

To construct a ZDD for the formula in Lemma 5, we need a ZDD for $\neg Cross(g)$. But note that the ZDD for the CNF $\neg Cross(g)$ is equal to the ZDD for the DNF $Cross(g)$ except replacing every positive literal p with the its negative literal $\neg p$ and vice-versa. Therefore, we want to swap p and $\neg p$ in $Cross(Z_g)$.

We retrieve the clauses with neither p nor $\neg p$ by

$$Z_1 = Subset0(Subset0(Cross(Z_g), p), \neg p).$$

Then we swap p with $\neg p$ in every clause where p appears positively:

$$Z_2 = Change(Subset1(Cross(Z_g), p), \neg p).$$

And we swap $\neg p$ with p in every clause where p appears negatively:

$$Z_3 = \mathit{Change}(\mathit{Subset1}(\mathit{Cross}(Z_g), \neg p), p).$$

Finally, taking the union of Z_1, Z_2 and Z_3 gives us the ZDD $\neg\mathit{Cross}(Z_g)$ encoding the CNF for the negation of $\mathit{Cross}(Z_g)$.

Let Z_y^+, Z_y^- and Z_y' be the ZDDs for f_y^+, f_y^- and f_y', respectively, constructed as described in Section 3.1. We compute the ZDDs for $(g \vee f_y^+)$ and $(\neg\mathit{Cross}(g) \vee f_y^-)$ by $\mathit{ClauseDistribution}(Z_g, Z_y^+)$ and $\mathit{ClauseDistribution}(\neg\mathit{Cross}(Z_g), Z_y^-)$, respectively. We then take the *Union* of these two ZDDs and Z_y' to get the ZDD for $(g \vee f_y^+) \wedge (\neg\mathit{Cross}(g) \vee f_y^-) \wedge f_y'$, which is exactly the ZDD for $f[y \mapsto g]$ by Lemma 5.

5 Experimental Evaluations

5.1 Experimental Methodology and and Setting

We perform a comparison between our ZDD-based synthesizer, ZSynth, and the tool RSynth described by [9], using challenging Π_2^P benchmarks from the QBFEVAL 2016 data set [18], the latest QBFEVAL set that includes a 2QBF (forall-exists) track, which is the format our benchmarks require. Each benchmark ran for 24 hours on Rice University's NOTS cluster with 64G RAM size. We focus our comparison on the Fixpoint Detection, MutexP, and QShifter benchmark families, omitting those families that are either too easy or too hard to solve for both tools, namely, the Tree, Ranking Functions, Reduction Finding, and Sorting Networks families [18]. For those families, either both tools solved all instances or none. Of these omitted benchmark families, Tree is very simple and is solved very quickly by both tools, while the others could be synthesized by neither tool. therefore we choose to focus on the three families that provide an interesting comparison. Fixpoint Detection, MutexP and QShifter have, respectively, 146, 7, and 6 instances.

For each tool we evaluate both total time and peak memory consumption for compilation, realizability, and synthesis, as well as the DD size for the original formula in each symbolic representation. We use the maximum cardinality search (MCS) heuristic [23] to determine the ordering of variables in both ZDDs and BDDs.Due to restrictions on available time and space resources, some benchmarks show out-of-time and out-of-memory failures. We measure the performance of both tools on the benchmarks that are solved. The experimental evaluations conclude that the ZDD-based approach is complementary to the BDD-based approach.

5.2 Compilation Time and Size of Diagram Representing Original Formula

We first compare the performance of CNF compilation into ZDDs and BDDs, following the first research question proposed in Section 1. The log-scale bar plot in

Fig. 1 presents compilation time for the benchmarks from the selection families, per Section 5.1. The size of the bars representing each formula is proportional to the compilation time.

The compilation into a ZDD takes polynomial (at most quadratic) time, because paths in the ZDD correspond to clauses, and therefore the size of the ZDD is always linear in the size of the formula. In contrast, the compilation into a BDD can be exponential, because paths in a BDD correspond to assignments, and therefore the number of paths can be exponential. The advantage of ZDDs as a compact representation is consistent with our conjecture. Across all benchmark families in QBFEVAL'16, compilation into ZDDs takes less time and space than BDDs in most cases.

It is worth noting that we construct here the ZDD representation of the CNF formulas by adding one clause at a time using the *Union* operator. Compilation could be further optimized by using a divide-and-conquer approach, where we split the set of clauses in half, construct ZDDs for each half recursively, and then take their union.

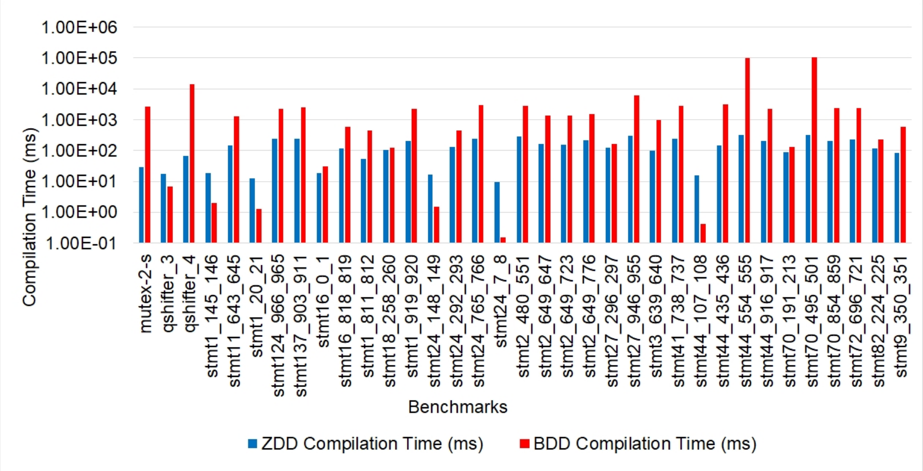


Fig. 1: Compilation time of the CNF: red = BDD, blue = ZDD

5.3 Realizability Time

The plot in Fig. 2 summarizes for each family the time spent on constructing the realizability set and checking partial and full realizability. The dashed lines in red illustrate RSynth results, while the solid lines in dark blue with the same shapes show ZSynth results. As each solvers have the families where it has an advantage in, we note how many instances of each family each solver is able to solve within a given time. We include data for all benchmarks that completed the

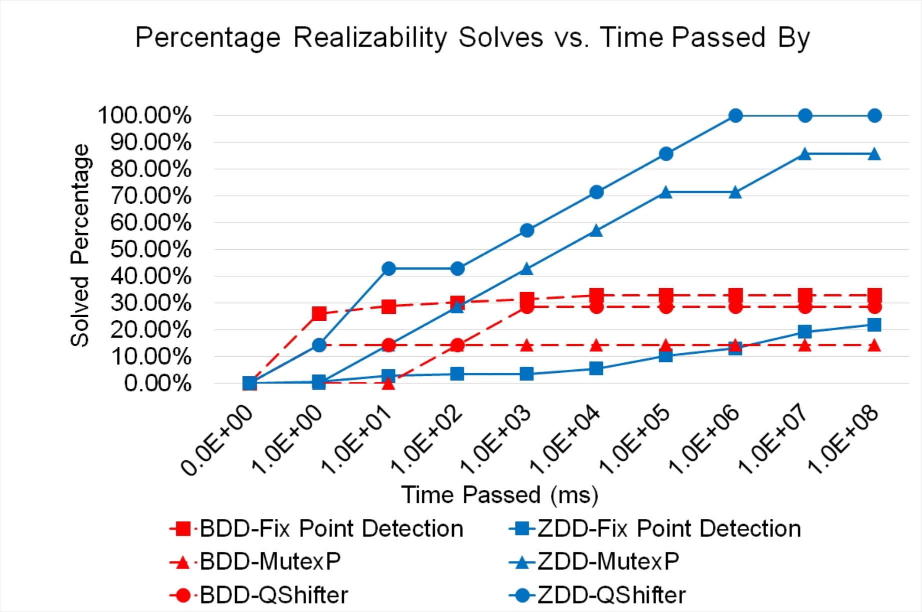


Fig. 2: Percentage solved for realizability within a given timeout. Dashed red = BDD, solid blue = ZDD.

realizability phase. The graph plots the percentage of benchmarks in each family that RSynth and ZSynth complete for a given timeout, with 100% meaning that all instances of that family were solved.

We see from Fig. 2 that RSynth solves more cases of the Fixpoint Detection family, and does so faster than ZSynth. Most of the cases it solves are completed in under 10ms. On the other hand, ZSynth has the advantage in the QShifter and MutexP families, for which it is able to solve more cases in a shorter time. Therefore, ZSynth and RSynth each performs better on different families of benchmarks. This allows us to answer the second research question proposed in Section 1 with the observation that neither approach dominates across the board, rather realizability performance is dependent on the benchmark family. As we see below in Section 5.4, these general results also extend to end-to-end synthesis.

5.4 End-to-End Time and Peak Memory

Our observations for end-to-end synthesis time–including compilation, realizability, and witness construction–are plotted in Fig. 3. Similarly to realizability time, the total end-to-end synthesis time shows strongly family-dependent results. Both ZSynth and RSynth display better relative performance on the same families as they did for realizability. In families where ZSynth solves more instances, including QShifter and MutexP, ZSynth also takes less time in most

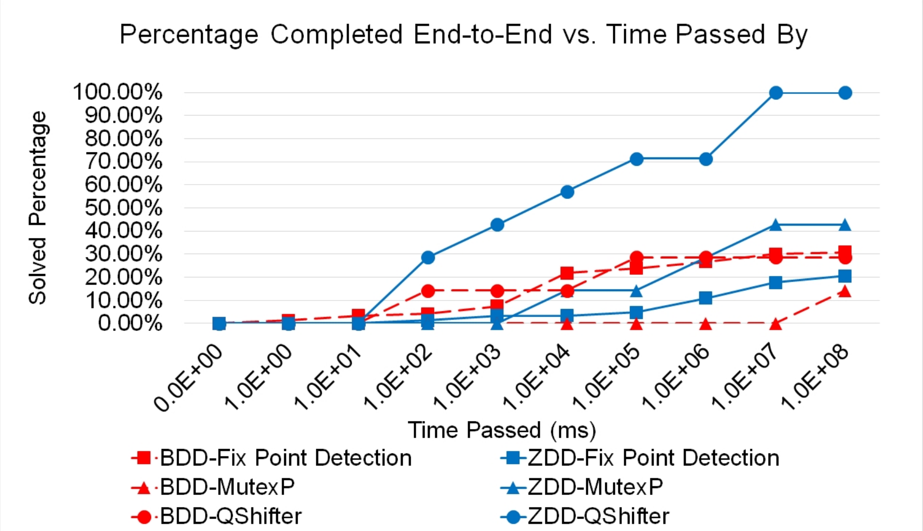


Fig. 3: Percentage solved end-to-end within a given timeout. Dashed red = BDD, solid blue = ZDD.

cases, and vice-versa for those families where RSynth solves more benchmarks end-to-end.

We observed in our experiments that memory and time were generally correlated, meaning that benchmarks that took more time also consumed more memory. This is expected when dealing with algorithms based on decision diagrams, since the biggest factor that impacts the performance of such algorithms is diagram size. In practice, memory comparison between RSynth and ZSynth in compilation, realizability and witness construction have similar patterns as the time comparison. Even if ZDDs have an advantage in representing the initial specification, the overall memory consumption for realizability and synthesis is, similarly to running time, largely dependent on the benchmark family.

5.5 Scalable Benchmarks Show ZDD has Slower Growing Demands of Time and Space

To analyze the scalability of ZDDs in relation to BDDs, as per the fifth research question in Section 1, we take a closer look at the running time and node counts of ZSynth and RSynth in the benchmarks of the QShifter family. All benchmarks in this family follow the same structure, just scaled based on a numerical parameter. For a parameter n, `qshifter_n` has 2^{2n+1} clauses, $2^n + n$ input variables and 2^n output variables, so we expect to see exponential trends in the measured values.

The results can be found in Fig. 4, which considers only QShifter because it can be scaled based on a parameter, and RSynth did not solve enough instances of MutexP to have an interesting scalability comparison. Since RSynth solves only up to the smallest instances in the QShifter family, we use the maximal time limit, illustrated by the "X" in the plot not connected to any line, as a conservative underestimation for the running time of further instances. (Therefore, the compilation, realizability and end-to-end times for RSynth in `qshifter_5` must be higher than the "X" mark.) As QShifter benchmarks are regular in their constructions, we can observe the trend of the exponent.

The results for RSynth, both for time and number of nodes, always has a steeper slope in the parameter n. Since the graph is in log scale, straight lines represent an exponential increase, and the slope represents the coefficient of the exponent. Therefore, although both ZSynth and RSynth grow exponentially, in both time and space, ZSynth is more efficient by an exponential factor.

These results suggest that there are families for which we can expect ZDD synthesizers to require significantly fewer resources in time and space as the size of the formulas grows. The QShifter family is one example of a family where the ZDD algorithm performs better by an exponential factor.

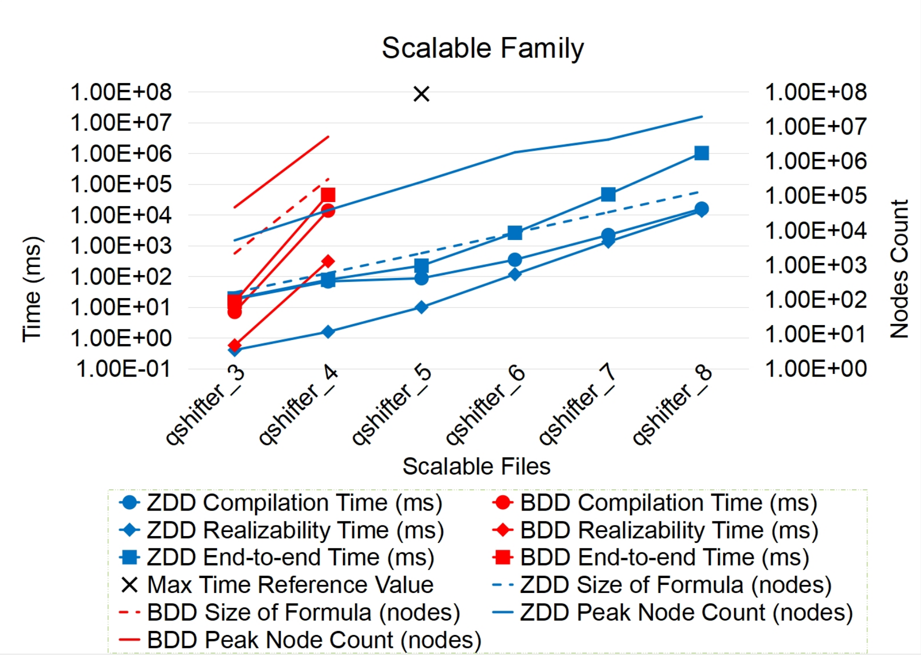


Fig. 4: Scalable family evaluations: dashed red = BDD, solid blue = ZDD.

5.6 Overall Comparison

As explained in Section 5.1, we focus on evaluating the synthesizers on the Fixpoint Detection, QShifter, and MutexP families of benchmarks [18]. ZSynth

Table 1: Percentage of end-to-end completed instances in each family.

Benchmark Family Name	RSynth (BDD)	ZSynth (ZDD)
Fixpoint Detection	30.82%	20.55%
MutexP	14.29%	42.86%
QShifter (scalable)	28.57%	100%

shows clear time and space advantages on the MutexP and QShifter families, while RSynth performs better in the Fixpoint Detection family. In Table 1, we show how much of each family either tool was able to solve.

Next, we summarize the overall results of our experimental performance comparison. In families where ZDD completed more instances end-to-end, we can see that ZDD has better performance in all bases of comparison, including compilation, realizability, and end-to-end time, as well as diagram node count for the original formula and peak node count. Additionally, Section 5.5 shows that there exist families of scalable benchmarks for which the time and space demands of ZDDs grow more slowly than BDDs by an exponential factor, as illustrated by the smaller slope in Fig. 4.

Even in the Fixpoint Detection family, where BDDs solve more instances, ZDDs show advantages in compilation time, initial diagram size, and smaller scaling slopes in time and space. In realizability and overall synthesis performance, neither our ZDD-based algorithm nor the BDD-based algorithm dominates across the board, each performing better in those families where it can solve more instances.

6 Conclusion

We conclude that ZDD-based algorithms are competitive with those based on BDDs, and both have their place in a portfolio of solvers for boolean synthesis. Since both BDDs and ZDDs can be converted to circuits, we advocate that an industrial solver would benefit from both approaches. In CNF-specified boolean-synthesis problems, BDD and ZDD are orthogonal approaches, and circumstances exist where each one of the solvers shows leading performance. For this type of problems, our portfolio advocates a multi-engine approach that is inclusive of both approaches.

As most tools for QBF solving and synthesis solving handle the input formula monolithically, future research based on this work includes an exploration of partitioning of variables [8] and factored synthesis [23] in the context of ZDDs. Another direction is to explore the usage of ZDD-based techniques in the context of temporal synthesis, cf. [24].

References

1. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about Boolean functional synthesis? In: Proc. 30th Int'l Conf. on Computer Aided Ver-

ification, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 251–269. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_14, https://doi.org/10.1007/978-3-319-96145-3_14

2. Bryant, R.: Graph-based algorithms for Boolean-function manipulation. IEEE Transactions on Computing **C-35**(8), 677–691 (1986)
3. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. In: Proc. 2018 IEEE Conf. on Formal Methods in Computer Aided Design. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603000
4. Chatalic, P., Simon, L.: Multi-resolution on compressed sets of clauses. In: Proc 12th IEEE Int'l Conf. on Tools with Artificial Intelligence. pp. 2–10. IEEE Computer Society (2000). https://doi.org/10.1109/TAI.2000.889839
5. Chatalic, P., Simon, L.: ZRES: The old Davis-Putman procedure meets ZBDD. In: Proc. 17th Int'l Conf. on Automated Deduction. Lecture Notes in Computer Science, vol. 1831, pp. 449–454. Springer (2000)
6. Crama, Y., Hammer, P.L.: Boolean functions: Theory, algorithms, and applications. Cambridge University Press (2011)
7. Dechter, R., van Beek, P.: Local and global relational consistency. Theor. Comput. Sci. **173**(1), 283–308 (1997)
8. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: Procount: Weighted projected model counting with graded project-join trees. In: Proc. 24th Int'l Conf. on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 12831, pp. 152–170. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_11
9. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based Boolean functional synthesis. In: Proc. 28th Int'l Conf. on Computer Aided Verification. Part II. Lecture Notes in Computer Science, vol. 9780, pp. 402–421. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_22
10. Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for Boolean function synthesis. In: Proc. 32nd Int'l Conf. on on Computer Aided Verification, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 611–633. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_31, https://doi.org/10.1007/978-3-030-53291-8_31
11. Gomes, C.P., Selman, B.: Algorithm portfolio design: Theory vs. practice. arXiv preprint arXiv:1302.1541 (2013)
12. Hofferek, G., Gupta, A., Könighofer, B., Jiang, J.R., Bloem, R.: Synthesizing multiple Boolean functions using interpolation on a single proof. In: Proc. 2013 IEEE Conf. Formal Methods in Computer-Aided Design. pp. 77–84. IEEE (2013)
13. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: Proc. 2015 IEEE Conf. Formal Methods in Computer-Aided Design. pp. 73–80. IEEE (2015)
14. Knuth, D.E.: The Art of Computer Programming, Volume 4, Pre-Fascicle 1B: A Draft of Section 7.1.4: Binary Decision Diagrams. Addison-Wesley Professional, 12th edn. (2009)
15. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: Proc. 12th Int'l Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 1855, pp. 113–123. Springer (2000). https://doi.org/10.1007/10722167_12
16. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. 30th Design Automation Conference. pp. 272–277. ACM Press (1993). https://doi.org/10.1145/157485.164890

17. Mishchenko, A.: Introduction to zero-suppressed decision diagrams. Synthesis Lectures on Digital Circuits and Systems **45** (2001)
18. Narizzano, M., Pulina, L., Tacchella, A.: The QBFEVAL web portal. In: Proc. 10th European Conf. on Logics in Artificial Intelligence. Lecture Notes in Computer Science, vol. 4160, pp. 494–497. Springer (2006). https://doi.org/10.1007/11853886_45, https://doi.org/10.1007/11853886_45
19. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Proc. 10th Int'l Conf. Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 3258, pp. 453–467. Springer (2004). https://doi.org/10.1007/978-3-540-30201-8_34
20. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Proc. 19th Int'l Conf. on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 9710, pp. 375–392. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_23
21. Sasao, T., Butler, J.T.: Applications of zero-suppressed decision diagrams. Synthesis Lectures on Digital Circuits and Systems **9**(2), 1–123 (2014)
22. Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. University of Colorado at Boulder (2015)
23. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: Proc. 2017 IEEE Conf. on Formal Methods in Computer Aided Design. pp. 124–131. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102250
24. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proc. 26th Int'l Joint Conf. on Artificial Intelligence. pp. 1362–1369. ijcai.org (2017)

Verification

Comparative Verification of the Digital Library of Mathematical Functions and Computer Algebra Systems

André Greiner-Petter[1] (✉), Howard S. Cohl[2], Abdou Youssef[2,3], Moritz Schubotz[1,4], Avi Trost[5], Rajen Dey[6], Akiko Aizawa[7], and Bela Gipp[1]

[1] University of Wuppertal, Wuppertal, Germany,
{greinerpetter,schubotz,gipp}@uni-wuppertal.de
[2] National Institute of Standards and Technology,
Mission Viejo, CA, U.S.A., howard.cohl@nist.gov
[3] George Washington University, Washington, D.C., U.S.A, ayoussef@gwu.edu
[4] FIZ Karlsruhe, Berlin, Germany, moritz.schubotz@fiz-karlsruhe.de
[5] Brown University, Providence, RI, U.S.A., avitrost@gmail.com
[6] University of California Berkeley, Berkeley, CA, U.S.A., rajhataj@gmail.com
[7] National Institute of Informatics, Tokyo, Japan, aizawa@nii.ac.jp

Abstract. Digital mathematical libraries assemble the knowledge of years of mathematical research. Numerous disciplines (e.g., physics, engineering, pure and applied mathematics) rely heavily on compendia gathered findings. Likewise, modern research applications rely more and more on computational solutions, which are often calculated and verified by computer algebra systems. Hence, the correctness, accuracy, and reliability of both digital mathematical libraries and computer algebra systems is a crucial attribute for modern research. In this paper, we present a novel approach to verify a digital mathematical library and two computer algebra systems with one another by converting mathematical expressions from one system to the other. We use our previously developed conversion tool (referred to as LaCASt) to translate formulae from the NIST Digital Library of Mathematical Functions to the computer algebra systems `Maple` and `Mathematica`. The contributions of our presented work are as follows: (1) we present the most comprehensive verification of computer algebra systems and digital mathematical libraries with one another; (2) we significantly enhance the performance of the underlying translator in terms of coverage and accuracy; and (3) we provide open access to translations for `Maple` and `Mathematica` of the formulae in the NIST Digital Library of Mathematical Functions.

Keywords: Presentation to Computation, LaCASt, LaTeX, Semantic LaTeX, Computer Algebra Systems, Digital Mathematical Library

1 Introduction

Digital Mathematical Libraries (DML) gather the knowledge and results from thousands of years of mathematical research. Even though pure and applied mathematics are precise disciplines, gathering their knowledge bases over many years results in

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 87–105, 2022.
https://doi.org/10.1007/978-3-030-99524-9_5

issues which every digital library shares: consistency, completeness, and accuracy. Likewise, Computer Algebra Systems (CAS)[8] play a crucial role in the modern era for pure and applied mathematics, and those fields which rely on them. CAS can be used to simplify, manipulate, compute, and visualize mathematical expressions. Accordingly, modern research regularly uses DML and CAS together. Nonetheless, DML [7,14] and CAS [1,20,11] are not exempt from having bugs or errors. Durán et al. [11] even raised the rather dramatic question: "*can we trust in [CAS]?*"

Existing comprehensive DML, such as the Digital Library of Mathematical Functions (DLMF) [10], are consistently updated and frequently corrected with errata[9]. Although each chapter of the DLMF has been carefully written, edited, validated, and proofread over many years, errors still remain. Maintaining a DML, such as the DLMF, is a laborious process. Likewise, CAS are eminently complex systems, and in the case of commercial products, often similar to black boxes in which the magic (i.e., the computations) happens in opaque private code [11]. CAS, especially commercial products, are often exclusively tested internally during development.

An independent examination process can improve testing and increase trust in the systems and libraries. Hence, we want to elaborate on the following research question.

> How can digital mathematical libraries and computer algebra systems be utilized to improve and verify one another?

Our initial approach for answering this question is inspired by our previous studies on translating DLMF equations to CAS [7]. In order to verify a translation tool from a specific LaTeX dialect to `Maple`[10]. , we performed *symbolic* and *numeric* evaluations on equations from the DLMF. Our approach presumes that a proven equation in a DML must be also valid in a CAS. In turn, a disparity in between the DML and CAS would lead to an issue in the translation process. However, assuming a correct translation, a disparity would also indicate an issue either in the DML source or the CAS implementation. In turn, we can take advantage of the same approach to improve and even verify DML with CAS and vice versa. Unfortunately, previous efforts to translate mathematical expressions from various formats, such as LaTeX [8,14,29], MathML [31], or OpenMath [18,30], to CAS syntax have shown that the translation will be the most critical part of this verification approach.

In this paper, we elaborate on the feasibility and limitations of the translation approach from DML to CAS as a possible answer to our research question. We further focus on the DLMF as our DML and the two general-purpose CAS `Maple` and `Mathematica` for this first study. This relatively sharp limitation is necessary in order to analyze the capabilities of the underlying approach to verify commercial CAS

[8] In the sequel, the acronyms CAS and DML are used, depending on the context, interchangeably with their plurals.

[9] https://dlmf.nist.gov/errata/ [accessed 09/01/2021]

[10] The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology, nor does it imply that the products so identified are necessarily the best available for the purpose. All trademarks mentioned herein belong to their respective owners.

and large DML. The DLMF uses semantic macros internally in order to disambiguate mathematical expressions [27,35]. These macros help to mitigate the open issue of retrieving sufficient semantic information from a context to perform translations to formal languages [31,14]. Further, the DLMF and general-purpose CAS have a relatively large overlap in coverage of special functions and orthogonal polynomials. Since many of those functions play a crucial role in a large variety of different research fields, we focus in this study mainly on these functions. Lastly, we will take our previously developed translation tool LACASt [8,14] as the baseline for translations from the DLMF to `Maple`. In this successor project, we focus on improving LACASt to minimize the negative effect of wrong translations as much as possible for our study. In the future, other DML and CAS can be improved and verified following the same approach by using a different translation approach depending on the data of the DML, e.g., MathML [31] or OpenMath [18].

In particular, in this paper, we fix the majority of the remaining issues of LACASt [7], which allows our tool to translate twice as many expressions from the DLMF to the CAS as before. Current extensions include the support for the mathematical operators: sum, product, limit, and integral, as well as overcoming semantic hurdles associated with Lagrange (prime) notations commonly used for differentiation. Further, we extend its support to include `Mathematica` using the freely available *Wolfram Engine for Developers* (WED)[11] (hereafter, with `Mathematica`, we refer to the WED). These improvements allow us to cover a larger portion of the DLMF, increase the reliability of the translations via LACASt, and allow for comparisons between two major general-purpose CAS for the first time, namely `Maple` and `Mathematica`. Finally, we provide open access to all the results contained within this paper, including all translations of DLMF formulae, an endpoint to LACASt[12], and the full source code of LACASt[13].

The paper is structured as follows. Section 2 explains the data in the DLMF. Section 3 focus on the improvements of LACASt that had been made to make the translation as comprehensive and reliable as possible for the upcoming evaluation. Section 4 explains the symbolic and numeric evaluation pipeline. Since Cohl et al. [7] only briefly sketched the approach of a numeric evaluation, we will provide an in-depth discussion of that process in Section 4. Subsequently, we analyze the results in Section 5. Finally, we conclude the findings and provide an outlook for upcoming projects in Section 6.

1.1 Related Work

Existing verification techniques for CAS often focus on specific subroutines or functions [26,20,5,12,6,25,21,17], such as a specific theorems [23], differential equations [19], or the implementation of the `math.h` library [24]. Most common are verification approaches that rely on intermediate verification languages [5,20,21,19,17], such as *Boogie* [25,2] or *Why3* [21,4], which, in turn, rely on proof assistants and theorem provers, such as *Coq* [5,3], *Isabelle* [19,28], or *HOL Light* [20,16,17]. Kaliszyk and Wiedijk [20] proposed on entire new CAS which is built on top of the proof assistant HOL Light so that each simplification step can be proven by the underlying

[11] https://www.wolfram.com/engine/ [accessed 09/01/2021]

[12] https://lacast.wmflabs.org/ [accessed 01/01/2022]

[13] https://github.com/ag-gipp/LaCASt [accessed 04/01/2022]

architecture. Lewis and Wester [26] manually compared the symbolic computations on polynomials and matrices with seven CAS. Aguirregabiria et al. [1] suggested to teach students the known traps and difficulties with evaluations in CAS instead to reduce the overreliance on computational solutions.

Cohl et al. [7] developed the aforementioned translation tool LaCASt, which translates expressions from a semantically enhanced LaTeX dialect to `Maple`. By evaluating the performance and accuracy of the translations, we were able to discover a sign-error in one the DLMF's equations [7]. While the evaluation was not intended to verify the DLMF, the translations by the rule-based translator LaCASt provided sufficient robustness to identify issues in the underlying library. To the best of our knowledge, besides this related evaluation via LaCASt, there are no existing libraries or tools that allow for automatic verification of DML.

2 The DLMF dataset

In the modern era, most mathematical texts (handbooks, journal publications, magazines, monographs, treatises, proceedings, etc.) are written using the document preparation system LaTeX. However, the focus of LaTeX is for precise control of the rendering mechanics rather than for a semantic description of its content. In contrast, CAS syntax is coercively unambiguous in order to interpret the input correctly. Hence, a transformation tool from DML to CAS must disambiguate mathematical expressions. While there is an ongoing effort towards such a process [32,22,34,13,36,33], there is no reliable tool available to disambiguate mathematics sufficiently to date.

The DLMF contains numerous relations between functions and many other properties. It is written in LaTeX but uses specific semantic macros when applicable [35]. These semantic macros represent a unique function or polynomial defined in the DLMF. Hence, the semantic LaTeX used in the DLMF is often unambiguous. For a successful evaluation via CAS, we also need to utilize all requirements of an equation, such as constraints, domains, or substitutions. The DLMF provides this additional data too and generally in a machine-readable form [35]. This data is accessible via the i-boxes (information boxes next to an equation marked with the icon ⓘ). If the information is not given in the attached i-box or the information is incorrect, the translation via LaCASt would fail. The i-boxes, however, do not contain information about branch cuts (see Section B) or constraints. Constraints are accessible if they are directly attached to an equation. If they appear in the text (or even a title), LaCASt cannot utilize them. The test dataset, we are using, was generated from DLMF Version 1.1.3 (2021-09-15) and contained 9,977 formulae with 1,505 defined symbols, 50,590 used symbols, 2,691 constraints, and 2,443 warnings for non-semantic expressions, i.e., expressions without semantic macros [35]. Note that the DLMF does not provide access to the underlying LaTeX source. Therefore, we added the source of every equation to our result dataset.

3 Semantic LaTeX to CAS translation

The aforementioned translator LaCASt was developed by Cohl and Greiner-Petter et al. [8,7,14]. They reported a coverage of 58.8% translations for a manually selected part of the DLMF to the CAS `Maple`. This version of LACAST serves as a baseline for our improvements. In order to verify their translations, they used symbolic and numeric evaluations and reported a success rate of ∼16% for symbolic and ∼12% for numeric verifications.

Evaluating the baseline on the entire DLMF result in a coverage of only 31.6%. Hence, we first want to increase the coverage of LACAST on the DLMF. To achieve this goal, we first increasing the number of translatable semantic macros by manually defining more translation patterns for special functions and orthogonal polynomials. For `Maple`, we increased the number from 201 to 261. For `Mathematica`, we define 279 new translation patterns which enables LACAST to perform translations to `Mathematica`. Even though the DLMF uses 675 distinguished semantic macros, we cover ∼70% of all DLMF equations with our extended list of translation patterns (see Zipf's law for mathematical notations [15]). In addition, we implemented rules for translations that are applicable in the context of the DLMF, e.g., ignore ellipsis following floating-point values or `\choose` always refers to a binomial expression. Finally, we tackle the remaining issues outlined by Cohl et al. [7] which can be categorized into three groups: (i) expressions of which the arguments of operators are not clear, namely sums, products, integrals, and limits; (ii) expressions with prime symbols indicating differentiation; and (iii) expressions that contain ellipsis. While we solve some of the cases in Group (iii) by ignoring ellipsis following floating-point values, most of these cases remain unresolved. In the following, we elaborate our solutions for (i) in Section 3.1 and (ii) in Section 3.2.

3.1 Parse sums, products, integrals, and limits

Here we consider common notations for the sum, product, integral, and limit operators. For these operators, one may consider mathematically essential operator metadata (MEOM). For all these operators, the MEOM includes *argument(s)* and *bound variable(s)*. The operators act on the arguments, which are themselves functions of the bound variable(s). For sums and products, the bound variables are referred to as *indices*. The bound variables for integrals[14] are called *integration variables*. For limits, the bound variables are continuous variables (for limits of continuous functions) and indices (for limits of sequences). For integrals, MEOM include precise descriptions of regions of integration (e.g., piecewise continuous paths/intervals/regions). For limits, MEOM include limit points (e.g., points in $\mathbb{R}^n$ or $\mathbb{C}^n$ for $n \in \mathbb{N}$), as well as information related to whether the limit to the limit point is independent or dependent on the direction in which the limit is taken (e.g., one-sided limits).

For a translation of mathematical expressions involving the LaTeX commands `\sum`, `\int`, `\prod`, and `\lim`, we must extract the MEOM. This is achieved by (a) determining the argument of the operator and (b) parsing corresponding subscripts, superscripts, and arguments. For integrals, the MEOM may be complicated, but certainly contains the argument (function which will be integrated), bound (integration) variable(s) and details related to the region of integration. Bound variable extraction is usually straightforward since it is usually contained within a differential expression

[14] The notion of integrals includes: antiderivatives (indefinite integrals), definite integrals, contour integrals, multiple (surface, volume, etc.) integrals, Riemannian volume integrals, Riemann integrals, Lebesgue integrals, Cauchy principal value integrals, etc.

(infinitesimal, pushforward, differential 1-form, exterior derivative, measure, etc.), e.g., $\mathrm{d}x$. Argument extraction is less straightforward since even though differential expressions are often given at the end of the argument, sometimes the differential expression appears in the numerator of a fraction (e.g., $\int \frac{f(x)\mathrm{d}x}{g(x)}$). In which case, the argument is everything to the right of the `\int` (neglecting its subscripts and superscripts) up to and including the fraction involving the differential expression (which may be replaced with 1). In cases where the differential expression is fully to the right of the argument, then it is a *termination symbol*. Note that some scientists use an alternate notation for integrals where the differential expression appears immediately to the right of the integral, e.g., $\int \mathrm{d}x f(x)$. However, this notation does not appear in the DLMF. If such notations are encountered, we follow the same approach that we used for sums, products, and limits (see Section 3.1).

Extraction of variables and corresponding MEOM The subscripts and superscripts of sums, products, limits, and integrals may be different for different notations and are therefore challenging to parse. For integrals, we extract the bound (integration) variable from the differential expression. For sums and products, the upper and lower bounds may appear in the subscript or superscript. Parsing subscripts is comparable with the problem of parsing constraints [7] (which are often not consistently formulated). We overcame this complexity by manually defining patterns of common constraints and refer to them as blueprints. This blueprint pattern approach allows LACAST to identify the MEOM in the sub- and superscripts. A more detailed explanations with examples about the blueprints is available in the Appendix A[15].

Identification of operator arguments Once we have extracted the bound variable for sums, products, and limits, we need to determine the end of the argument. We analyzed all sums in the DLMF and developed a heuristic that covers all the formulae in the DLMF and potentially a large portion of general mathematics. Let x be the extracted bound variable. For sums, we consider a summand as a part of the argument if (I) it is the very first summand after the operation; or (II) x is an element of the current summand; or (III) x is an element of the following summand (subsequent to the current summand) and there is no termination symbol between the current summand and the summand which contains x with an equal or lower depth according to the parse tree (i.e., closer to the root). We consider a summand as a single logical construct since addition and subtraction are granted a lower operator precedence than multiplication in mathematical expressions. Similarly, parentheses are granted higher precedence and, thus, a sequence wrapped in parentheses is part of the argument if it obeys the rules (I-III). Summands, and such sequences, are always entirely part of sums, products, and limits or entirely not.

A termination symbol always marks the end of the argument list. Termination symbols are relation symbols, e.g., $=$, $\neq$, $\leq$, closing parentheses or brackets, e.g.,),], or $>$, and other operators with MEOMs, if and only if, they define the same bound variable. If x is part of a subsequent operation, then the following operator

[15] The Appendix is available at https://arxiv.org/abs/2201.09488.

is considered as part of the argument (as in (II)). However, a special condition for termination symbols is that it is only a termination symbol for the current chain of arguments. Consider a sum over a fraction of sums. In that case, we may reach a termination symbol within the fraction. However, the termination symbol would be deeper inside the parse tree as compared to the current list of arguments. Hence, we used the depth to determine if a termination symbol should be recognized or not. Consider an unusual notation with the binomial coefficient as an example

$$\sum_{k=0}^{n}\binom{n}{k} = \sum_{k=0}^{n}\frac{\prod_{m=1}^{n}m}{\prod_{m=1}^{k}m\prod_{m=1}^{n-k}m}. \tag{1}$$

This equation contains two termination symbols, marked red and green. The red termination symbol $=$ is obviously for the first sum on the left-hand side of the equation. The green termination symbol $\prod$ terminates the product to the left because both products run over the same bound variable m. In addition, none of the other $=$ signs are termination symbols for the sum on the right-hand side of the equation because they are deeper in the parse tree and thus do not terminate the sum.

Note that `varN` in the blueprints also matches multiple bound variable, e.g., $\sum_{m,k\in A}$. In such cases, x from above is a list of bound variables and a summand is part of the argument if one of the elements of x is within this summand. Due to the translation, the operation will be split into two preceding operations, i.e., $\sum_{m,k\in A}$ becomes $\sum_{m\in A}\sum_{k\in A}$. Figure 1 shows the extracted arguments for some example sums. The same rules apply for extraction of arguments for products and limits.

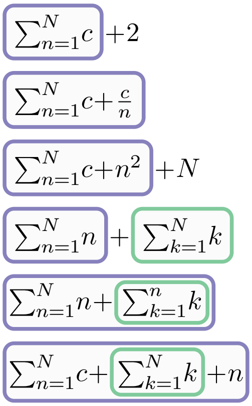

$\sum_{n=1}^{N}c+2$
$\sum_{n=1}^{N}c+\frac{c}{n}$
$\sum_{n=1}^{N}c+n^2+N$
$\sum_{n=1}^{N}n+\sum_{k=1}^{N}k$
$\sum_{n=1}^{N}n+\sum_{k=1}^{n}k$
$\sum_{n=1}^{N}c+\sum_{k=1}^{N}k+n$

Fig. 1: Example argument identifications for sums.

3.2 Lagrange's notation for differentiation and derivatives

Another remaining issue is the Lagrange (prime) notation for differentiation, since it does not outwardly provide sufficient semantic information. This notation presents two challenges. First, we do not know with respect to which variable the differentiation should be performed. Consider for example the Hurwitz zeta function $\zeta(s,a)$ [10, §25.11]. In the case of a differentiation $\zeta'(s,a)$, it is not clear if the function should be differentiated with respect to s or a. To remedy this issue, we analyzed all formulae in the DLMF which use prime notations and determined which variables (slots) for which functions represent the variables of the differentiation. Based on our analysis, we extended the translation patterns by meta information for semantic macros according to the slot of differentiation. For instance, in the case of the Hurwitz zeta function, the first slot is the slot for prime differentiation, i.e., $\zeta'(s,a)=\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s,a)$. The identified variables of differentiations for the special functions in the DLMF can be considered to be the standard slots of differentiations, e.g., in other DML, $\zeta'(s,a)$ most likely refers to $\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s,a)$.

The second challenge occurs if the slot of differentiation contains complex expressions rather than single symbols, e.g., $\zeta'(s^2,a)$. In this case, $\zeta'(s^2,a) = \frac{\mathrm{d}}{\mathrm{d}(s^2)}\zeta(s^2,a)$ instead of $\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s^2,a)$. Since CAS often do not support derivatives with respect to complex expressions, we use the inbuilt substitution functions[16] in the CAS to overcome this issue. To do so, we use a temporary variable `temp` for the substitution. CAS perform substitutions from the inside to the outside. Hence, we can use the same temporary variable `temp` even for nested substitutions. Table 1 shows the translation performed for $\zeta'(s^2,a)$. CAS may provide optional arguments to calculate the derivatives for certain special functions, e.g., `Zeta(n,z,a)` in `Maple` for the n-th derivative of the Hurwitz zeta function. However, this shorthand notation is generally not supported (e.g., `Mathematica` does not define such an optional parameter). Our substitution approach is more lengthy but also more reliable. Unfortunately, lengthy expressions generally harm the performance of CAS, especially for symbolic manipulations. Hence, we have a genuine interest in keeping translations short, straightforward and readable. Thus, the substitution translation pattern is only triggered if the variable of differentiation is not a single identifier. Note that this substitution only triggers on semantic macros. Generic functions, including prime notations, are still skipped.

A related problem to MEOM of sums, products, integrals, limits, and differentiations are the notations of derivatives. The semantic macro for derivatives `\deriv{w}{x}` (rendered as $\frac{dw}{dx}$) is often used with an empty first argument to render the function behind the derivative notation, e.g., `\deriv{}{x}\sin@{x}` for $\frac{\mathrm{d}}{\mathrm{d}x}\sin x$. This leads to the same problem we faced above for identifying MEOMs. In this case, we use the same heuristic as we did for sums, products, and limits. Note that derivatives may be written following the function argument, e.g., $\sin(x)\frac{\mathrm{d}}{\mathrm{d}x}$. If we are unable to identify any following summand that contains the variable of differentiation before we reach a termination symbol, we look for arguments prior to the derivative according to the heuristic (I-III).

Table 1: Example translations for the prime derivative of the Hurwitz zeta function with respect to s^2.

System	$\zeta'(s^2,a)$
DLMF	`\Hurwitzzeta'@{s^2}{a}`

`Maple`	`subs(temp=(s)^(2),diff(`
	`Zeta(0,temp,a),temp$(1)))`

`Mathematica`	`D[HurwitzZeta[temp,a],`
	`{temp,1}]/.temp->(s)^(2)`

Wronskians With the support of prime differentiation described above, we are also able to translate the Wronskian [10, (1.13.4)] to `Maple` and `Mathematica`. A translation requires one to identify the variable of differentiation from the elements of the Wronskian, e.g., z for $\mathscr{W}\{\mathrm{Ai}(z),\mathrm{Bi}(z)\}$ from [10, (9.2.7)]. We analyzed all Wronskians in the DLMF and discovered that most Wronskians have a special

[16] Note that `Maple` also support an evaluation substitution via the two-argument `eval` function. Since our substitution only triggers on semantic macros, we only use `subs` if the function is defined in `Maple`. In turn, as far as we know, there is no practical difference between `subs` and the two-argument `eval` in our case.

function in its argument—such as the example above. Hence, we can use our previously inserted metadata information about the slots of differentiation to extract the variable of differentiation from the semantic macros. If the semantic macro argument is a complex expression, we search for the identifier in the arguments that appear in both elements of the Wronskian. For example, in $\mathscr{W}\{\mathrm{Ai}(z^a),\zeta(z^2,a)\}$, we extract z as the variable since it is the only identifier that appears in the arguments z^a and z^2 of the elements. This approach is also used when there is no semantic macro involved, i.e., from $\mathscr{W}\{z^a,z^2\}$ we extract z as well. If LaCASt extracts multiple candidates or none, it throws a translation exception.

4 Evaluation of the DLMF using CAS

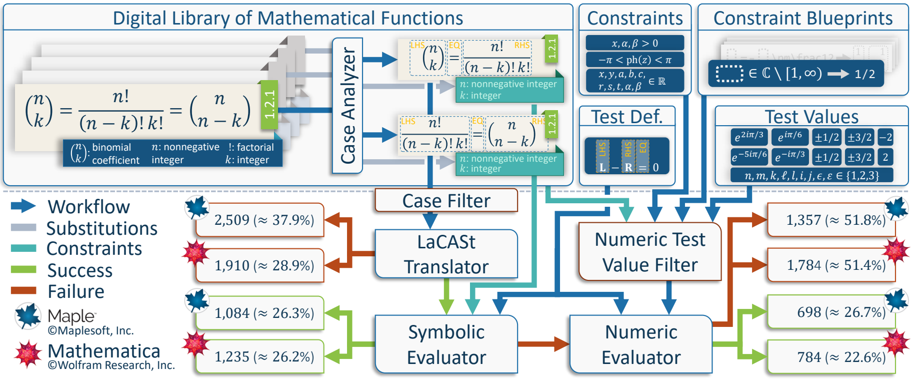


Fig. 2: The workflow of the evaluation engine and the overall results. Errors and abortions are not included. The generated dataset contains 9,977 equations. In total, the case analyzer splits the data into 10,930 cases of which 4,307 cases were filtered. This sums up to a set of 6,623 test cases in total.

For evaluating the DLMF with `Maple` and `Mathematica`, we follow the same approach as demonstrated in [7], i.e., we symbolically and numerically verify the equations in the DLMF with CAS. If a verification fails, symbolically and numerically, we identified an issue either in the DLMF, the CAS, or the verification pipeline. Note that an issue does not necessarily represent errors/bugs in the DLMF, CAS, or LaCASt (see the discussion about branch cuts in Section B). Figure 2 illustrates the pipeline of the evaluation engine. First, we analyze every equation in the DLMF (hereafter referred to as test cases). A case analyzer splits multiple relations in a single line into multiple test cases. Note that only the adjacent relations are considered, i.e., with $f(z)=g(z)=h(z)$, we generate two test cases $f(z)=g(z)$ and $g(z)=h(z)$ but not $f(z)=h(z)$. In addition, expressions with ± and ∓ are split accordingly, e.g., $i^{\pm i}=\mathrm{e}^{\mp\pi/2}$ [10, (4.4.12)] is split into $i^{+i}=\mathrm{e}^{-\pi/2}$ and $i^{-i}=\mathrm{e}^{+\pi/2}$. The analyzer utilizes the attached additional information in each line, i.e., the URL in the DLMF, the used and defined symbols, and the constraints. If a used symbol is defined elsewhere in the DLMF, it performs substitutions. For example, the multi-equation [10, (9.6.2)] is split into six test cases and every ζ is replaced by $\frac{2}{3}z^{3/2}$ as defined in [10, (9.6.1)]. The substitution is performed on the parse tree of expressions [14]. A definition is only considered as such, if the defining symbol is identical to the equation's left-hand side. That means, $z = (\frac{3}{2}\zeta)^{3/2}$ [10, (9.6.10)] is not considered as a definition for ζ. Further, semantic macros are never substituted by their definitions. Translations for semantic macros are exclusively defined by the authors. For example, the equation [10, (11.5.2)] contains the Struve $\mathbf{K}_\nu(z)$ function. Since `Mathematica` does not contain this function, we defined an alternative translation to its definition $\mathbf{H}_\nu(z) - Y_\nu(z)$ in [10, (11.2.5)] with the Struve function $\mathbf{H}_\nu(z)$ and the Bessel function of the second kind $Y_\nu(z)$, because both of these functions are supported by `Mathematica`. The second entry in Table 3 in the Appendix D shows the translation for this test case.

Next, the analyzer checks for additional constraints defined by the used symbols recursively. The mentioned Struve $\mathbf{K}_\nu(z)$ test case [10, (11.5.2)] contains the Gamma function. Since the definition of the Gamma function [10, (5.2.1)] has a constraint $\Re z > 0$, the numeric evaluation must respect this constraint too. For this purpose, the case analyzer first tries to link the variables in constraints to the arguments of the functions. For example, the constraint $\Re z > 0$ sets a constraint for the first argument z of the Gamma function. Next, we check all arguments in the actual test case at the same position. The test case contains $\Gamma(\nu + 1/2)$. In turn, the variable z in the constraint of the definition of the Gamma function $\Re z > 0$ is replaced by the actual argument used in the test case. This adds the constraint $\Re(\nu + 1/2) > 0$ to the test case. This process is performed recursively. If a constraint does not contain any variable that is used in the final test case, the constraint is dropped.

In total, the case analyzer would identify four additional constraints for the test case [10, (11.5.2)]. Table 3 in the Appendix D shows the applied constraints (including the directly attached constraint $\Re z > 0$ and the manually defined global constraints from Figure 3). Note that the constraints may contain variables that do not appear in the actual test case, such as $\Re \nu + k + 1 > 0$. Such constraints do not have any effect on the evaluation because if a constraint cannot be computed to true or false, the constraint is ignored. Unfortunately, this recursive loading of additional constraints may generate impossible conditions in certain cases, such as $|\Gamma(iy)|$ [10, (5.4.3)]. There are no valid real values of y such that $\Re(iy) > 0$. In turn, every test value would be filtered out, and the numeric evaluation would not verify the equation. However, such cases are the minority and we were able to increase the number of correct evaluations with this feature.

To avoid a large portion of incorrect calculations, the analyzer filters the dataset before translating the test cases. We apply two filter rules to the case analyzer. First, we filter expressions that do not contain any semantic macros. Due to the limitations of LaCASt, these expressions most likely result in wrong translations. Further, it filters out several meaningless expressions that are not verifiable, such as $z = x$ in [10, (4.2.4)]. The result dataset flag these cases with '*Skipped - no semantic math*'. Note that the result dataset still contains the translations for these cases to provide a complete picture of the DLMF. Second, we filter expressions that contain ellipsis[17] (e.g., `\cdots`), approximations, and asymptotics (e.g., $\mathcal{O}(z^2)$) since those expressions cannot be evaluated with the proposed approach. Further, a definition is skipped if it is not a definition of a semantic macro, such as [10, (2.3.13)], because definitions without

[17] Note that we filter out ellipsis (e.g., `\cdots`) but not single dots (e.g., `\cdot`).

an appropriate counterpart in the CAS are meaningless to evaluate. Definitions of semantic macros, on the other hand, are of special interest and remain in the test set since they allow us to test if a function in the CAS obeys the actual mathematical definition in the DLMF. If the case analyzer (see Figure 2) is unable to detect a relation, i.e., split an expression on $<$, $\leq$, $\geq$, $>$, $=$, or $\neq$, the line in the dataset is also skipped because the evaluation approach relies on relations to test. After splitting multi-equations (e.g., $\pm$, $\mp$, $a=b=c$), filtering out all non-semantic expressions, non-semantic macro definitions, ellipsis, approximations, and asymptotics, we end up with 6,623 test cases in total from the entire DLMF.

After generating the test case with all constraints, we translate the expression to the CAS representation. Every successfully translated test case is then symbolically verified, i.e., the CAS tries to simplify the difference of an equation to zero. Non-equation relations simplifies to Booleans. Non-simplified expressions are verified numerically for manually defined test values, i.e., we calculate actual numeric values for both sides of an equation and check their equivalence.

4.1 Symbolic Evaluation

The symbolic evaluation was performed for `Maple` as in [7]. However, we use the newer version `Maple` 2020. Another feature we added to LaCASt is the support of packages in `Maple`. Some functions are only available in modules (packages) that must be preloaded, such as `QPochhammer` in the package `QDifferenceEquations`[18]. The general `simplify` method in `Maple` does not cover q-hypergeometric functions. Hence, whenever LaCASt loads functions from the q-hyper-geometric package, the better performing `QSimplify` method is used. With the WED and the new support for `Mathematica` in LaCASt, we perform the symbolic and numeric tests for `Mathematica` as well. The symbolic evaluation in `Mathematica` relies on the full simplification[19]. For `Maple` and `Mathematica`, we defined the global assumptions $x,y\in\mathbb{R}$ and $k,n,m\in\mathbb{N}$. Constraints of test cases are added to their assumptions to support simplification. Adding more global assumptions for symbolic computation generally harms the performance since CAS internally uses assumptions for simplifications. It turned out that by adding more custom assumptions, the number of successfully simplified expressions decreases.

4.2 Numerical Evaluation

Defining an accurate test set of values to analyze an equivalence can be an arbitrarily complex process. It would make sense that every expression is tested on specific values according to the containing functions. However, this laborious process is not suitable for evaluating the entire DML and CAS. It makes more sense to develop a general set of test values that (i) generally covers interesting domains and (ii) avoid singularities, branch cuts, and similar problematic regions. Considering these two attributes, we come up with the ten test points illustrated in Figure 3. It contains four complex values on the unit circle and six points on the real axis. The test values cover the

[18] https://jp.maplesoft.com/support/help/Maple/view.aspx?path=QDifferenceEquations/QPochhammer [accessed 09/01/2021]

[19] https://reference.wolfram.com/language/ref/FullSimplify.html [accessed 09/01/2021]

general area of interest (complex values in all four quadrants, negative and positive real values) and avoid the typical singularities at $\{0,\pm 1,\pm i\}$. In addition, several variables are tied to specific values for entire sections. Hence, we applied additional global constraints to the test cases.

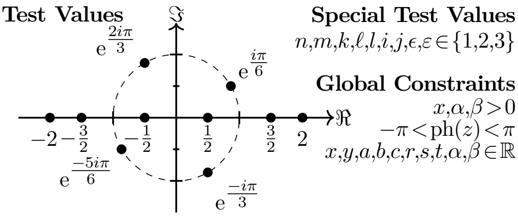


Fig. 3: The ten numeric test values in the complex plane for general variables. The dashed line represents the unit circle $|z| = 1$. At the right, we show the set of values for special variable values and general global constraints. On the right, i is referring to a generic variable and not to the imaginary unit.

The numeric evaluation engine heavily relies on the performance of extracting free variables from an expression. Unfortunately, the inbuilt functions in CAS, if available, are not very reliable. As the authors explained in [7], a custom algorithm within `Maple` was necessary to extract identifiers. `Mathematica` has the undocumented function `Reduce‘FreeVariables` for this purpose. However, both systems, the custom solution in `Maple` and the inbuilt `Mathematica` function, have problems distinguishing free variables of entire expressions from the bound variables in MEOMs, e.g., integration and continuous variables. `Mathematica` sometimes does not extract a variable but returns the unevaluated input instead. We regularly faced this issue for integrals. However, we discovered one example without integrals. For `EulerE[n,0]` from [10, (24.4.26)], we expected to extract $\{n\}$ as the set of free variables but instead received a set of the unevaluated expression itself $\{$`EulerE[n,0]`$\}$[20]. Since the extended version of LaCASt handles operators, including bound variables of MEOMs, we drop the use of internal methods in CAS and extend LaCASt to extract identifiers from an expression. During a translation process, LaCASt tags every single identifier as a variable, as long as it is not an element of a MEOM. This simple approach proves to be very efficient since it is implemented alongside the translation process itself and is already more powerful as compared to the existing inbuilt CAS solutions. We defined subscripts of identifiers as a part of the identifier, e.g., z_1 and z_2 are extracted as variables from $z_1 + z_2$ rather than z.

The general pipeline for a numeric evaluation works as follows. First, we replace all substitutions and extract the variables from the left- and right-hand sides of the test expression via LaCASt. For the previously mentioned example of the Struve function [10, (11.5.2)], LaCASt identifies two variables in the expression, ν and z. According to the values in Figure 3, ν and z are set to the general ten values. A numeric test contains every combination of test values for all variables. Hence, we generate 100 test calculations for [10, (11.5.2)]. Afterward, we filter the test values that violate the attached constraints. In the case of the Struve function, we end up with 25 test cases.

In addition, we apply a limit of 300 calculations for each test case and abort a computation after 30 seconds due to computational limitations. If the test case generates more than 300 test values, only the first 300 are used. Finally, we calculate

[20] The bug was reported to and confirmed by Wolfram Research Version 12.0.

the result for every remaining test value, i.e., we replace every variable by their value and calculate the result. The replacement is done by `Mathematica`'s `ReplaceAll` method because the more appropriate method `With`, for unknown reasons, does not always replace all variables by their values. We wrap test expressions in `Normal` for numeric evaluations to avoid conditional expressions, which may cause incorrect calculations (see Section 5.1 for a more detailed discussion of conditional outputs). After replacing variables by their values, we trigger numeric computation. If the absolute value of the result (i.e., the difference between left- and right-hand side of the equation) is below the defined threshold of 0.001 or true (in the case of inequalities), the test calculation is considered successful. A numeric test case is only considered successful if and only if every test calculation was successful. If a numeric test case fails, we store the information on which values it failed and how many of these were successful.

5 Results

The translations to `Maple` and `Mathematica`, the symbolic results, the numeric computations, and an overview PDF of the reported bugs to `Mathematica` are available online on our demopage. In the following, we mainly focus on `Mathematica` because of page limitations and because `Maple` has been investigated more closely by [7]. The results for `Maple` are also available online. Compared to the baseline ($\approx$31%), our improvements doubled the amount translations ($\approx$62%) for `Maple` and reach $\approx$71% for `Mathematica`. The majority of expressions that cannot be translated contain macros that have no adequate translation pattern to the CAS, such as the macros for interval Weierstrass lattice roots [10, §23.3(i)] and the multivariate hypergeometric function [10, (19.16.9)]. Other errors (6% for `Maple` and `Mathematica`) occur for several reasons. For example, out of the 418 errors in translations to `Mathematica`, 130 caused an error because the MEOM of an operator could not be extracted, 86 contained prime notations that do not refer to differentiations, 92 failed because of the underlying LaTeX parser [34], and in 46 cases, the arguments of a DLMF macro could not be extracted.

Out of 4,713 translated expressions, 1,235 (26.2%) were successfully simplified by `Mathematica` (1,084 of 4,114 or 26.3% in `Maple`). For `Mathematica`, we also count results that are equal to 0 under certain conditions as successful (called `ConditionalExpression`). We identified 65 of these conditional results: 15 of the conditions are equal to constraints that were provided in the surrounding text but not in the info box of the DLMF equation; 30 were produced due to branch cut issues (see Section B in the Appendix); and 20 were the same as attached in the DLMF but reformulated, e.g., $z \in \mathbb{C}\backslash(1,\infty)$ from [10, (25.12.2)] was reformulated to $\Im z \neq 0 \vee \Re z < 1$. The remaining translated but not symbolically verified expressions were numerically evaluated for the test values in Figure 3. For the 3,474 cases, 784 (22.6%) were successfully verified numerically by `Mathematica` (698 of 2,618 or 26.7% by `Maple`[21]). For 1,784 the numeric evaluation failed. In the evaluation process, 655

[21] Due to computational issues, 120 cases must have been skipped manually. 292 cases resulted in an error during symbolic verification and, therefore, were skipped also for numeric evaluations. Considering these skipped cases as failures, decreases the numerically verified cases to 23% in `Maple`.

computations timed out and 180 failed due to errors in `Mathematica`. Of the 1,784 failed cases, 691 failed partially, i.e., there was at least one successful calculation among the tested values. For 1,091 all test values failed. Table 3 in the Appendix D shows the results for three sample test cases. The first case is a false positive evaluation because of a wrong translation. The second case is valid, but the numeric evaluation failed due to a bug in Mathematica (see next subsection). The last example is valid and was verified numerically but was too complex for symbolic verifications.

5.1 Error Analysis

The numeric tests' performance strongly depends on the correct attached and utilized information. The first example in Table 3 in the Appendix D illustrates the difficulty of the task on a relatively easy case. Here, the argument of f was not explicitly given, such as in $f(x)$. Hence, LACAST translated f as a variable. Unfortunately, this resulted in a false verification symbolically and numerically. This type of error mostly appears in the first three chapters of the DLMF because they use generic functions frequently. We hoped to skip such cases by filtering expressions without semantic macros. Unfortunately, this derivative notation uses the semantic macro `deriv`. In the future, we filter expressions that contain semantic macros that are not linked to a special function or orthogonal polynomial.

As an attempt to investigate the reliability of the numeric test pipeline, we can run numeric evaluations on symbolically verified test cases. Since `Mathematica` already approved a translation symbolically, the numeric test should be successful if the pipeline is reliable. Of the 1,235 symbolically successful tests, only 94 (7.6%) failed numerically. None of the failed test cases failed entirely, i.e., for every test case, at least one test value was verified. Manually investigating the failed cases reveal 74 cases that failed due to an `Indeterminate` response from `Mathematica` and 5 returned `infinity`, which clearly indicates that the tested numeric values were invalid, e.g., due to testing on singularities. Of the remaining 15 cases, two were identical: [10, (15.9.2)] and [10, (18.5.9)]. This reduces the remaining failed cases to 14. We evaluated invalid values for 12 of these because the constraints for the values were given in the surrounding text but not in the info boxes. The remaining 2 cases revealed a bug in `Mathematica` regarding conditional outputs (see below). The results indicate that the numeric test pipeline is reliable, at least for relatively simple cases that were previously symbolically verified. The main reason for the high number of failed numerical cases in the entire DLMF (1,784) are due to missing constraints in the i-boxes and branch cut issues (see Section B in the Appendix), i.e., we evaluated expressions on invalid values.

Bug reports `Mathematica` has trouble with certain integrals, which, by default, generate conditional outputs if applicable. With the method `Normal`, we can suppress conditional outputs. However, it only hides the condition rather than evaluating the expression to a non-conditional output. For example, integral expressions in [10, (10.9.1)] are automatically evaluated to the Bessel function $J_0(|z|)$ for the condition[22] $z \in \mathbb{R}$ rather than $J_0(z)$ for all $z \in \mathbb{C}$. Setting the `GenerateConditions`[23] option

[22] $J_0(x)$ with $x \in \mathbb{R}$ is even. Hence, $J_0(|z|)$ is correct under the given condition.

[23] https://reference.wolfram.com/language/ref/GenerateConditions.html [accessed 09/01/2021]

to `None` does not change the output. `Normal` only hides $z \in \mathbb{R}$ but still returns $J_0(|z|)$. To fix this issue, for example in (10.9.1) and (10.9.4), we are forced to set `GenerateConditions` to false.

Setting `GenerateConditions` to false, on the other hand, reveals severe errors in several other cases. Consider $\int_z^\infty t^{-1} \mathrm{e}^{-t} \mathrm{d}t$ [10, (8.4.4)], which gets evaluated to $\Gamma(0,z)$ but (condition) for $\Re z > 0 \wedge \Im z = 0$. With `GenerateConditions` set to false, the integral incorrectly evaluates to $\Gamma(0,z)+\ln(z)$. This happened with the 2 cases mentioned above. With the same setting, the difference of the left- and right-hand sides of [10, (10.43.8)] is evaluated to 0.398942 for $x, \nu = 1.5$. If we evaluate the same expression on x, $\nu = \frac{3}{2}$ the result is `Indeterminate` due to `infinity`. For this issue, one may use `NIntegrate` rather than `Integrate` to compute the integral. However, evaluating via `NIntegrate` decreases the number of successful numeric evaluations in general. We have revealed errors with conditional outputs in (8.4.4), (10.22.39), (10.43.8-10), and (11.5.2) (in [10]). In addition, we identified one critical error in `Mathematica`. For [10, (18.17.47)], WED (`Mathematica`'s kernel) ran into a *segmentation fault (core dumped)* for $n > 1$. The kernel of the full version of `Mathematica` gracefully died without returning an output[24].

Besides `Mathematica`, we also identified several issues in the DLMF. None of the newly identified issues were critical, such as the reported sign error from the previous project [7], but generally refer to missing or wrong attached semantic information. With the generated results, we can effectively fix these errors and further semantically enhance the DLMF. For example, some definitions are not marked as such, e.g., $Q(z) = \int_0^\infty \mathrm{e}^{-zt} q(t) \mathrm{d}t$ [10, (2.4.2)]. In [10, (10.24.4)], ν must be a real value but was linked to a *complex parameter* and x should be positive real. An entire group of cases [10, (10.19.10-11)] also discovered the incorrect use of semantic macros. In these formulae, $P_k(a)$ and $Q_k(a)$ are defined but had been incorrectly marked up as Legendre functions going all the way back to DLMF Version 1.0.0 (May 7, 2010). In some cases, equations are mistakenly marked as definitions, e.g., [10, (9.10.10)] and [10, (9.13.1)] are annotated as local definitions of n. We also identified an error in LaCASt, which incorrectly translated the exponential integrals $E_1(z)$, $\mathrm{Ei}(x)$ and $\mathrm{Ein}(z)$ (defined in [10, §6.2(i)]). A more explanatory overview of discovered, reported, and fixed issues in the DLMF, `Mathematica`, and `Maple` is provided in the Appendix C.

6 Conclusion

We have presented a novel approach to verify the theoretical digital mathematical library DLMF with the power of two major general-purpose computer algebra systems `Maple` and `Mathematica`. With LaCASt, we transformed the semantically enhanced LaTeX expressions from the DLMF to each CAS. Afterward, we symbolically and numerically evaluated the DLMF expressions in each CAS. Our results are auspicious and provide useful information to maintain and extend the DLMF efficiently. We further identified several errors in `Mathematica`, `Maple` [7], the DLMF, and the transformation tool LaCASt, proving the profit of the presented verification approach.

[24] All errors were reported to and partially confirmed by Wolfram Research. See Appendix C for more information.

Further, we provide open access to all results, including translations and evaluations[25]. and to the source code of LaCASt[26].

The presented results show a promising step towards an answer for our initial research question. By translating an equation from a DML to a CAS, automatic verifications of that equation in the CAS allows us to detect issues in either the DML source or the CAS implementation. Each analyzed failed verification successively improves the DML or the CAS. Further, analyzing a large number of equations from the DML may be used to finally verify a CAS. In addition, the approach can be extended to cover other DML and CAS by exploiting different translation approaches, e.g., via MathML [31] or OpenMath [18].

Nonetheless, the analysis of the results, especially for an entire DML, is cumbersome. Minor missing semantic information, e.g., a missing constraint or not respected branch cut positions, leads to a relatively large number of false positives, i.e., unverified expressions correct in the DML and the CAS. This makes a generalization of the approach challenging because all semantics of an equation must be taken into account for a trustworthy evaluation. Furthermore, evaluating equations on a small number of discrete values will never provide sufficient confidence to verify a formula, which leads to an unpredictable number of true negatives, i.e., erroneous equations that pass all tests. A more sophisticated selection of critical values or other numeric tools with automatic results verification (such as variants of Newton's interval method) potentially mitigates this issue in the future. After all, we conclude that the approach provides valuable information to complement, improve, and maintain the DLMF, `Maple`, and `Mathematica`. A trustworthy verification, on the other hand, might be out of reach.

6.1 Future Work

The resulting dataset provides valuable information about the differences between CAS and the DLMF. These differences had not been largely studied in the past and are worthy of analysis. Especially a comprehensive and machine-readable list of branch cut positioning in different systems is a desired goal [9]. Hence, we will continue to work closely together with the editors of the DLMF to improve further and expand the available information on the DLMF. Finally, the numeric evaluation approach would benefit from test values dependent on the actual functions involved. For example, the current layout of the test values was designed to avoid problematic regions, such as branch cuts. However, for identifying differences in the DLMF and CAS, especially for analyzing the positioning of branch cuts, an automatic evaluation of these particular values would be very beneficial and can be used to collect a comprehensive, inter-system library of branch cuts. Therefore, we will further study the possibility of linking semantic macros with numeric regions of interest.

Acknowledgements We thank Jürgen Gerhard from Maplesoft for providing access and support for Maple. We also thank the DLMF editors for their assistance and support. This work was supported by the German Research Foundation (DFG grant no.: GI 1259/1) and the German Academic Exchange Service (DAAD grant no.: 57515245).

[25] `https://lacast.wmflabs.org` [accessed 01/01/2022]

[26] `https://github.com/ag-gipp/LaCASt` [accessed 04/01/2022]

References

1. Aguirregabiria, J.M., Hernández, A.M., Rivas, M.: Are we careful enough when using computer algebra? Computers in Physics **8**(1), 56–61 (1994). https://doi.org/10.1063/1.4823260
2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, pp. 364–387. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11804192_17
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq´Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer Berlin Heidelberg (2004)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. Boogie 2011: First International Workshop on Intermediate Verification Languages pp. 53–64 (5 2011), `https://hal.inria.fr/hal-00790310/document`
5. Boulmé, S., Hardin, T., Hirschkoff, D., Ménissier-Morain, V., Rioboo, R.: On the way to certify computer algebra systems. Electronic Notes in Theoretical Computer Science **23**(3), 370–385 (1999). https://doi.org/10.1016/S1571-0661(05)80609-7, cALCULEMUS 99, Systems for Integrated Computation and Deduction (associated to FLoC'99, the 1999 Federated Logic Conference)
6. Carette, J., Kucera, M.: Partial evaluation of Maple. Science of Computer Programming **76**(6), 469–491 (6 2011). https://doi.org/10.1016/j.scico.2010.12.001
7. Cohl, H.S., Greiner-Petter, A., Schubotz, M.: Automated symbolic and numerical testing of DLMF formulae using computer algebra systems. In: Intelligent Computer Mathematics CICM. vol. 11006, pp. 39–52. Springer (2018). https://doi.org/10.1007/978-3-319-96812-4_4
8. Cohl, H.S., Schubotz, M., Youssef, A., Greiner-Petter, A., Gerhard, J., Saunders, B.V., McClain, M.A., Bang, J., Chen, K.: Semantic preserving bijective mappings of mathematical formulae between document preparation systems and computer algebra systems. In: Intelligent Computer Mathematics CICM. pp. 115–131. Springer (2017). https://doi.org/10.1007/978-3-319-62075-6_9
9. Corless, R.M., Jeffrey, D.J., Watt, S.M., Davenport, J.H.: "According to Abramowitz and Stegun" or arccoth needn't be uncouth. SIGSAM Bulletin **34**(2), 58–65 (2000). https://doi.org/10.1145/362001.362023
10. DLMF: *NIST Digital Library of Mathematical Functions*. https://dlmf.nist.gov/, Release 1.1.4 of 2022-01-15, F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds.
11. Durán, A.J., Pérez, M., Varona, J.L.: The misfortunes of a trio of mathematicians using computer algebra systems. Can we trust in them? Notices of the AMS **61**(10), 1249–1252 (2014)
12. Elphick, D., Leuschel, M., Cox, S.: Partial evaluation of MATLAB. In: Gen. Prog. and Component Eng., pp. 344–363. Springer (2003). https://doi.org/10.1007/978-3-540-39815-8_21
13. Greiner-Petter, A., Schubotz, M., Aizawa, A., Gipp, B.: Making presentation math computable: Proposing a context sensitive approach for translating LaTeX to computer algebra systems. In: International Congress of Mathematical Software (ICMS). Lecture Notes in Computer Science, vol. 12097, pp. 335–341. Springer (2020). https://doi.org/10.1007/978-3-030-52200-1_33

14. Greiner-Petter, A., Schubotz, M., Cohl, H.S., Gipp, B.: Semantic preserving bijective mappings for expressions involving special functions between computer algebra systems and document preparation systems. Aslib Journal of Information Management **71**(3), 415–439 (2019). https://doi.org/10.1108/AJIM-08-2018-0185
15. Greiner-Petter, A., Schubotz, M., Müller, F., Breitinger, C., Cohl, H.S., Aizawa, A., Gipp, B.: Discovering mathematical objects of interest - A study of mathematical notations. In: WWW. pp. 1445–1456. ACM / IW3C2 (2020). https://doi.org/10.1145/3366423.3380218
16. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) Formal Methods in Computer-Aided Design (FMCAD). Lecture Notes in Computer Science, vol. 1166, pp. 265–269. Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0031814
17. Harrison, J.R., Théry, L.: A skeptic's approach to combining HOL and Maple **21**(3), 279–294. https://doi.org/10.1023/A:1006023127567
18. Heras, J., Pascual, V., Rubio, J.: Using open mathematical documents to interface computer algebra and proof assistant systems. In: Intelligent Computer Mathematics MKM at CICM. Lecture Notes in Computer Science, vol. 5625, pp. 467–473. Springer (2009). https://doi.org/10.1007/978-3-642-02614-0_37
19. Hickman, T., Laursen, C.P., Foster, S.: Certifying differential equation solutions from computer algebra systems in Isabelle/HOL `http://arxiv.org/abs/2102.02679`
20. Kaliszyk, C., Wiedijk, F.: Certified computer algebra on top of an interactive theorem prover. In: Towards Mechanized Math. Assist., pp. 94–105. Springer (2007). https://doi.org/10.1007/978-3-540-73086-6_8
21. Khan, M.T.: Formal Specification and Verification of Computer Algebra Software. phdthesis, Johannes Kepler University Linz (Apr 2014)
22. Kristianto, G.Y., Topić, G., Aizawa, A.: Utilizing dependency relationships between math expressions in math IR. Information Retrieval Journal **20**(2), 132–167 (3 2017). https://doi.org/10.1007/s10791-017-9296-8
23. Lambán, L., Rubio, J., Martín-Mateos, F.J., Ruiz-Reina, J.L.: Verifying the bridge between simplicial topology and algebra: the Eilenberg-Zilber algorithm. Logic Journal of IGPL **22**(1), 39–65 (8 2013). https://doi.org/10.1093/jigpal/jzt034
24. Lee, W., Sharma, R., Aiken, A.: On automatically proving the correctness of math.h implementations. Proc. ACM on Prog. Lang. (POPL) **2**(47), 1–32 (2018). https://doi.org/10.1145/3158135
25. Leino, K.R.M.: Program proving using intermediate verification languages (IVLs) like Boogie and Why3. ACM SIGAda Ada Letters **32**(3), 25–26 (11 2012). https://doi.org/10.1145/2402709.2402689
26. Lewis, R.H., Wester, M.: Comparison of polynomial-oriented computer algebra systems. SIGSAM Bull. **33**(4), 5–13 (12 1999). https://doi.org/10.1145/500457.500459
27. Miller, B.R., Youssef, A.: Technical aspects of the digital library of mathematical functions. Ann. Math. Artif. Intell. **38**(1-3), 121–136 (2003). https://doi.org/10.1023/A:1022967814992
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer Berlin Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
29. Parisse, B.: Compiling LATEX to computer algebra-enabled HTML5 `http://arxiv.org/abs/1707.01271`
30. Prieto, H., Dalmas, S., Papegay, Y.: Mathematica as an OpenMath application **34**(2), 22–26. https://doi.org/10.1145/362001.362016
31. Schubotz, M., Greiner-Petter, A., Scharpf, P., Meuschke, N., Cohl, H.S., Gipp, B.: Improving the representation and conversion of mathematical formulae by considering their textual context. In: ACM/IEEE JCDL. pp. 233–242. ACM (2018). https://doi.org/10.1145/3197026.3197058

32. Schubotz, M., Grigorev, A., Leich, M., Cohl, H.S., Meuschke, N., Gipp, B., Youssef, A.S., Markl, V.: Semantification of identifiers in mathematics for better math information retrieval. In: ACM SIGIR'16. pp. 135–144. ACM Press (2016). https://doi.org/10.1145/2911451.2911503
33. Shan, R., Youssef, A.: Towards math terms disambiguation using machine learning. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) Proceedings of the International Conference on Intelligent Computer Mathematics (CICM). Lecture Notes in Computer Science, vol. 12833, pp. 90–106. Springer. https://doi.org/10.1007/978-3-030-81097-9_7
34. Youssef, A.: Part-of-math tagging and applications. In: Intelligent Computer Mathematics CICM. Lecture Notes in Computer Science, vol. 10383, pp. 356–374. Springer (2017). https://doi.org/10.1007/978-3-319-62075-6_25
35. Youssef, A., Miller, B.R.: A contextual and labeled math-dataset derived from NIST's DLMF. In: Intelligent Computer Mathematics CICM. Lecture Notes in Computer Science, vol. 12236, pp. 324–330. Springer (2020). https://doi.org/10.1007/978-3-030-53518-6_25
36. Zanibbi, R., Oard, D.W., Agarwal, A., Mansouri, B.: Overview of ARQMath 2020: CLEF lab on answer retrieval for questions on math. In: CLEF. Lecture Notes in Computer Science, vol. 12260, pp. 169–193. Springer (2020). https://doi.org/10.1007/978-3-030-58219-7_15

Verifying Fortran Programs with CIVL

Wenhao Wu[1]✉, Jan Hückelheim[2], Paul D. Hovland[2], and Stephen F. Siegel[1]

[1] University of Delaware, Newark DE 19716, USA
{wuwenhao, siegel}@udel.edu
[2] Argonne National Laboratory, Lemont IL 60439, USA
{jhueckelheim, hovland}@anl.gov

Abstract. Fortran is widely used in computational science, engineering, and high performance computing. This paper presents an extension to the CIVL verification framework to check correctness properties of Fortran programs. Unlike previous work that translates Fortran to C, LLVM IR, or other intermediate formats before verification, our work allows CIVL to directly consume Fortran source files. We extended the parsing, translation, and analysis phases to support Fortran-specific features such as array slicing and reshaping, and to find program violations that are specific to Fortran, such as argument aliasing rule violations, invalid use of variable and function attributes, or defects due to Fortran's unspecified expression evaluation order. We demonstrate the usefulness of our tool on a verification benchmark suite and kernels extracted from a real world application.

Keywords: Fortran · verification · static analysis · model checking

1 Introduction

Fortran is a structured imperative programming language with a unique set of features, such as common data blocks and array reshaping and sectioning, that support efficient numerical computing. Many scientific applications, especially those requiring high performance, are written entirely in Fortran; others have core subroutines or rely on external components written in Fortran. A 2018 report from the European Performance Optimisation and Productivity Centre states that over half of the 151 HPC programs the centre had analyzed over a two-year period were written in pure Fortran or a combination of Fortran and C or C++ [10]. Likewise, 12 of the 33 HPC benchmark applications in the U.S. Department of Energy's widely-used CORAL suite have components written in Fortran [17].

The Fortran language has been used and revised for decades, and it has had many standard versions. Early versions of Fortran employed a fixed-form coding style, well-suited for punch cards and with strict positional constraints. Beginning with Fortran 90, a free-form style was introduced, enabling more structured programs, eliminating limits on line lengths, and providing more flexibility with

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 106–124, 2022.
https://doi.org/10.1007/978-3-030-99524-9_6

character positioning by removing the restriction that the first six columns could be used only for labels and continuation characters. Modern Fortran programs tend to use the free-form style, but programs derived from a Fortran 77 predecessor or relying on legacy components may rely on fixed-form style or a mix of both styles.

Fortran is used to implement applications such as Nek5000 [21] or Flash [32] that are used for critical tasks such as nuclear reactor licensing reviews or to answer important scientific questions. These applications are often computationally demanding, requiring hours of computation on millions of execution units. Because of the critical importance and high resource requirements of these applications, one would like to verify their correctness.

The Fortran language itself provides little support for verification—not even assertions. Compilers can check certain simple syntactic and semantic properties, and static analyzers such as Coverity [35] can detect standard violations and other anomalies. But there are very few tools that can be used to specify and verify deeper functional correctness properties of programs, and nothing like the rich ecosystem of formal verification tools for C.

One might approach Fortran program verification by using a source-to-source translator such as f2c [11] to convert to C and then applying a C verifier. Unfortunately, even if the translator provides a completely valid translation, defects in the original code may not be preserved in the translated code; an example is given in Section 3.1. In addition, the C verifier may not be able to access translator support libraries, or defects that manifest themselves via the library may be difficult to map back to the original program.

A second approach is to use a compiler front end to convert Fortran code into an intermediate form such as the LLVM [16] Intermediate Representation (IR), and then apply a verifier for the IR. This is more difficult than it appears: most verifiers that consume LLVM IR are tuned to a specific source language and front end and cannot be easily modified to effectively verify multiple languages. This issue is explored in [13] in the case of SMACK, a C-via-LLVM verifier that has been extended to provide limited support for other languages, including Fortran. Moreover, as with source-to-source translators, the front end may translate away a defect in the original program; this is discussed in Section 3.2.

In this paper, we present an approach to extending the CIVL [33] verification framework so that it can be directly applied to Fortran source code. CIVL is a model checker that uses symbolic execution to verify correctness properties and was originally designed for programs written in C with a set of parallel programming language extensions such as OpenMP [26]. In our extended framework, summarized in Section 2, a new Fortran front end with a static analyzer has been integrated into the system. In Section 3 we describe the sequence of defect-preserving transformations that convert the Fortran source to the CIVL intermediate verification language, CIVL-C. Proper handling of arrays is a special concern, discussed in Section 4. The Fortran extension supports a subset of the major features defined in the language standard, focused on those features necessary to verify code excerpts from real world applications.

In Section 5, we evaluate our approach by verifying several examples of Fortran code, including (1) a custom Fortran benchmark suite designed to test CIVL's ability to verify programs using unique Fortran features such as array slicing and reshaping, (2) a published micro verification benchmark [13], and (3) a set of code excerpts from Nek5000 [21]. The evaluation employs both of CIVL's verification modes on Fortran programs. The first uses assumptions and assertions inserted in the program to specify the desired correctness properties. The second compares two programs with the same input-ouput signatures to determine whether they are functionally equivalent.

Related work is discussed in Section 6, and conclusions and future work are summarized in Section 7.

2 Overview of CIVL Extension

The Concurrency Intermediate Verification Language (CIVL) platform was developed to verify C programs that use various concurrency language extensions [33]. CIVL has two primary components: a front end and a back end verifier. The front end consumes a set of source files, which, prior to this work, had to be written in C or CUDA-C, possibly using certain CIVL extensions to C. These source files may use one or more concurrency language extensions, including MPI [18], OpenMP [26], Pthreads [25], and CUDA-C [24]. The input is parsed, analyzed and merged to create a single abstract syntax tree (AST) representing the whole program. This AST then undergoes a sequence of transformations to replace all of the concurrency primitives with equivalent CIVL-C primitives, and to simplify the AST in other ways, resulting in a "pure" CIVL-C AST.

The back end first converts the pure AST to a lower-level representation in which each procedure is represented as a program graph. A node in this graph represents a program counter value, i.e., a location in the procedure body. An edge represents an atomic transition, and is decorated with a guard expression that specifies when the transition is enabled, and a basic statement, such as an assignment. The verifier then performs an explicit enumeration of the reachable states of the program ("model checking"). This is carried out by depth-first search, while saving the seen states in a hash table. Each state maps variables to symbolic expressions and includes a path condition—a symbolic expression of boolean type that records the guards that held along the explored path ("symbolic execution"). An interleaving model of concurrency is used, and processes can be created and destroyed dynamically.

During the search, automated theorem provers are invoked to determine whether the path condition has become unsatisfiable (in which case the search backtracks) and to check assertions. CIVL checks both explicit assertions appearing in the program and implicit assertions (a divisor is not 0, a pointer deference is valid, and so on). The supported provers include Z3 [20], CVC4 [4], Why3 [5], and a number of additional provers invoked by Why3.

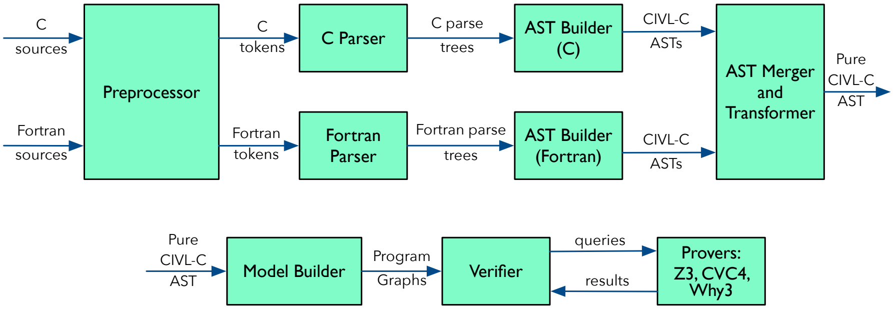


Fig. 1. CIVL architecture: front end (top) and back end (bottom)

Figure 1 shows the tool prior to this work and highlights the extensions developed as part of this paper. Modifications were made to both the front and back end to enable the direct application of CIVL to Fortran source code.

The CIVL preprocessor was generalized to accept a superset of C and Fortran: it is common practice to use C preprocessor directives in Fortran programs and Fortran compilers can invoke the preprocessor as a first pass. The tokens emanating from the preprocessor have a type specific to the source language—C or Fortran—which is determined by the file suffix or a command line option. It is possible to invoke CIVL on a mix of C and Fortran source files—each will be preprocessed separately and yield a separate stream of tokens in the correct language.

Each Fortran token stream enters the Fortran parser. This was produced by the parser generator ANTLR [28] using a grammar derived from the Open Fortran Project (OFP) [31]. We extended the grammar by adding support for CIVL primitives, such as assertions and assumptions, which can appear as structured comments in the Fortran source. The parser produces a parse tree, which is then converted to a CIVL-C AST. Each C token stream follows a similar path, and also results in a CIVL-C AST. Finally, the individual ASTs, together with ASTs generated from any libraries, are merged into a single AST, analogous to the linking phase in a standard compilation flow. The supported Fortran subset is listed below:

- **program units**: main programs, subroutines, and functions
- **statements**: allocate, assignment, call, computed goto, data, dimension, do, exit, goto, if, implicit, intent, parameter, pointer assignment, print, return, stop, target, type declaration, and write
- **expressions**: variable references, function calls, operators for scalar types
- **intrinsic functions**: mod, max, abs, sin, cos, atan, and sqrt
- **extended features**: CIVL preprocessor directives and CIVL primitives.

The transformation from a Fortran parse tree to a CIVL-C AST is quite involved because the languages differ substantially. In almost every case, we were able to find a way to represent a Fortran statement—in a semantics-preserving and defect-preserving way—using existing CIVL-C AST nodes; in a few cases we had to add new fields to the AST node. Issues include the Fortran "intent" specification for a procedure parameter (in, out, or in/out); pass-by-reference semantics; and advanced array operations. Details for some of these translations are described in Section 3.

The verifier was also upgraded to check specific Fortran runtime constraints during state exploration. For example, the verifier normally uses short-circuit semantics for evaluating *and* and *or* expressions. This is appropriate for C, but Fortran does not mandate short-circuiting or the order in which subexpressions are evaluated. Since evaluation can result in error, a verifier which assumes short-circuiting semantics could miss defects in a Fortran program. By default, our modified verifier turns off short-circuiting for Fortran code.

3 Defect-Preserving Translation

When used for verification, it is crucial that all translation phases preserve defects. This is in contrast to translation and lowering phases in a compiler, which generally are allowed to narrow the semantics of a program or choose arbitrarily from multiple interpretations. In this section, we first demonstrate with small examples that an approach relying on existing source-to-source translation tools such as f2c, or compiler front ends such as Flang, is bound to miss certain defects in the Fortran input, since these defects are removed by these tools. One might be tempted to argue that defects which disappear during translation or compilation are not really important. However, these defects are still present in the original source code and may manifest themselves when a different compiler is used or when other seemingly innocent changes are made to the code or the translation/compilation tool chain.

3.1 Translation from Source to Source

Figure 2 shows a procedure in Fortran 77 and its C translation produced by f2c. The example extracts a value x from an array at the given index, and computes $\max(x, 0)$. An array bounds check is performed in the same boolean expression in which the array is accessed. The C code is certainly valid, because it uses short-circuiting when evaluating logic expressions. Thus, the evaluation of the second part of the boolean expression is skipped if the first part is false. Fortran, on the other hand, does not define the order in which the subexpressions are evaluated,

```
if (idx .le. size_arr .and.
    arr(idx) .ge. 0) then
  relu = arr(idx)
else
  relu = 0
end if
```

```
/* Function Body */
if (*idx <= *arr_size__ && arr[*idx] >= 0.f) {
  *relu = arr[*idx];
} else {
  *relu = 0.f;
}
```

Fig. 2. Applying f2c to Fortran *and* operator removes a defect

and the compiler may choose an evaluation order that causes an out-of-bounds access in the second half of the expression. The implementation-defined order chosen by f2c happens to remove this defect during translation, which makes it difficult to detect for a verifier that is only provided with the C program. Nevertheless, the Fortran program may break when a different translator or compiler tool chain is used to execute it.

Besides the lack of defect preservation, there are other drawbacks when using a source-to-source converter, including the fact that some of them introduce hard-to-verify external headers or libraries to simulate Fortran behaviors. Further, by verifying translated code, source file information (e.g., file and identifier names, code locations, etc.) can be harder to communicate to the user, and translation tools may actually introduce new errors, leading to another potential source of unreliable verification results.

3.2 Translation for Compilation

A popular approach for verifying source code is to build a verifier based on a mature compiler tool chain (e.g., LLVM [16]). This allows verification researchers to spend more of their time on research and less time on maintaining language front ends, and allows robust support of a variety of languages. We argue that such an approach, while also very valuable, achieves a different outcome than what we present in our work. Compiler front ends such as Clang or Flang are not developed with the goal of preserving defects, and defective programs may be lowered into correct LLVM intermediate representation (IR). Furthermore, the compiler may in rare cases introduce new defects due to compiler bugs. In the absence of such compiler bugs, verification based on the IR will ensure that the input program is correct *if compiled with the same compiler and settings that were used for verification.* With our approach, we instead aim to verify that a program adheres to the language standard.

Figure 3 shows the LLVM-IR produced by Flang (version 1.5 2017-05-01) for the Fortran code snippet in Figure 2. Similar to the case with f2c, it first checks the array bounds by comparing %15 (element index) with %17 (array size). If the index is out of bounds (i.e., %18 is evaluated as *true*), then the control flow skips the block that accesses the array elements, and the second subexpression

```
L.LB1_339: ; preds = %L.entry
%14 = bitcast i64* %idx to i32*, !dbg !18
%15 = load i32, i32* %14, align 4, !dbg !18
%16 = bitcast i64* %arr_size to i32*, !dbg !18
%17 = load i32, i32* %16, align 4, !dbg !18
%18 = icmp sgt i32 %15, %17, !dbg !18
br i1 %18, label %L.LB1_313, label %L.LB1_349, !dbg !18
..
L.LB1_313: ; preds = %L.LB1_349, %L.LB1_339
%41 = bitcast i64* %relu to float*, !dbg !21
store float 0.000000e+00, float* %41, align 4, !dbg !21
br label %L.LB1_314
```

Fig. 3. Result of applying Flang to Fortran code of Figure 2

```
subroutine intent_bad(i)
  integer, intent(out) :: i
  i = i + 1
end subroutine
```

```
subroutine intent_good(i)
  integer, intent(inout) :: i
  i = i + 1
end subroutine
```

```
void INCR(int* __OUT_I) {
  int I;
  I = I + 1;
  *__OUT_I = I;
}
```

Fig. 4. Fortran routine that fails to conform to specified intent; one that conforms; and CIVL translation of the non-conforming code

in the condition expression is omitted. This means that the defect in the original Fortran code is undetectable in the IR.

Figure 4 is another case where Flang translates an incorrect program into valid IR. A Fortran subroutine may use the `INTENT` attribute in an illegal way, for example by declaring an argument as `INTENT`(`OUT`) and subsequently reading from it. This is problematic since the value of such a variable is undefined at the entry of the subroutine, even if it was initialized in the caller. Flang nevertheless generates identical LLVM IR for the two subroutines, the first of which violates the Fortran standard, the second of which declares the same argument as `INTENT`(`INOUT`) and hence correctly passes the variable into and out of the subroutine.

3.3 Translation for Verification

Based on these observations, we extended CIVL with a front end to translate Fortran to CIVL-C ASTs in a way that is designed to preserve defects. The front end avoids AST simplifications and optimizations that may introduce or remove defects, or that may hide violations of the Fortran language standard. The short-circuit evaluation of logic expressions is disabled by default when verifying Fortran source. When processing the code of Figure 2, the CIVL-C AST builder thus keeps both subexpressions in the condition, and all parts of the expression are evaluated in the verification phase. The model checker consequently reports an out-of-bounds access in the array.[3]

We also developed a static analyzer to detect certain defects before the program is even translated to a CIVL-C AST. The analyzer mainly checks constraints on variable attributes or procedure specifications. For example, variables in Fortran may have the `ALLOCATABLE`, `POINTER`, or `TARGET` attribute. It is legal to pointer-assign a variable with the `POINTER` attribute to a variable with the `TARGET` or `POINTER` attribute, but not to a variable without any of these attributes. Both sides of each pointer assignment are statically checked for required attributes by the analyzer. When all constraints of a specific attribute are verified for each associated variable, that attribute information is not passed to the model checker. Similarly, a subroutine or function is only allowed to be recursively called if it has the `RECURSIVE` attribute, and our analyzer checks this by searching for loops in the call graph and checking if subroutines or functions

[3] It is also possible (however unlikely) that a defect may manifest only when short-circuiting is *enabled*. A strictly conservative solution could use nondeterministic choice to decide, at each logical expression, whether to short-circuit. We plan to add such an option to CIVL.

that are part of a loop have the required attribute. As a result, this kind of constraint is checked by the analyzer and it is not necessary to include certain attributes in the CIVL-C AST.

The defect-preserving translation is mainly performed by the Fortran AST builder shown in Figure 1. For properties that can not be verified by the analyzer, the translation phase inserts auxiliary structures into the CIVL-C AST for verification in a later phase. For example, the subroutines in Figure 4 have distinct CIVL-C AST structures. A formal parameter having INTENT(OUT) attribute is initialized with a value representing "undefined." This allows the model checker to find and report a violation (reading an undefined value) during the transition executing the assignment statement. The CIVL translation of the incorrect routine is shown in Figure 4(right).

In summary, our extended front end focuses on preserving defects and translates source code into a CIVL-C AST specifically designed for verification. Violations of variable attributes and function specifications are guaranteed by performing specialized analysis or by inserting auxiliary information into the AST that is analyzed in a later phase.

4 Fortran Array Modeling

Fortran arrays are more powerful than arrays in most other languages, and require special handling during the translation to CIVL-C. Section 4.1 will briefly discuss some of the features of Fortran arrays, before we discuss how these features are modeled in Section 4.2.

4.1 Fortran Array Semantics

Fortran natively supports multi-dimensional arrays. For example, b and c in Figure 5 are two-dimensional arrays. Fortran stores arrays in column major style, unlike C arrays, which are stored in row major style.

```
REAL:: b(6,3), c(0:9,-3:3), u(3)
REAL, POINTER, DIMENSION(:) :: p
INTEGER, DIMENSION(3) :: idx
! copy columns -1, 0, 1 from every other row of c into the first 5 rows of b
b(1:5,:) = c(::2,-1:1)
! fill the array idx with constant values 1, 4, 17
idx = (/1, 4, 17/)
! use the array idx as indices into a. This will copy a[1,4,17] into u[1,2,3]
u = a(idx)
! associate the pointer p with column 1 in array c
p => c(:,1)
b = 42.0
```

Fig. 5. Examples of Fortran array usage: a 2-dimensional array of size 6 × 3, a 2-dimensional array with non-default index ranges, a pointer to a one-dimensional array, and two one-dimensional arrays of size 3, one for integers and one for reals, are declared. Following that, several data copy operations and pointer associations are performed.

Arrays in Fortran are 1-based by default, just like in Matlab or Julia, but unlike in C and many other languages. However, Fortran allows the base to be specified for each array dimension. For example, `c` in Figure 5 represents a two dimensional array whose row dimension of size 10 is 0-based and whose column dimension ranges from -3 to 3. Array sizes and index ranges can be either defined statically or calculated from parameters or function and subroutine arguments.

Fortran programs can in most situations determine the size of arrays using the intrinsic `size` or `shape` functions. It is also possible to modify an entire array, or an array along an entire dimension, without explicitly referring to its size. For example, one can assign a scalar value to an entire array though a simple assignment as shown in line 12 of Figure 5. Fortran compilers usually implement this behavior using an array descriptor that is embedded in the generated program and contains the array size and shape information.

Furthermore, Fortran supports the extraction of slices from an array by specifying a *subscript triplet* for each dimension, which specifies a lower and upper bound on the index as well as a stride. It is possible to omit the lower (and/or upper) bound, in which case the start (and/or end) of the array is used. An optional stride n can be specified to extract only every n-th element. For example, line 5 in Figure 5 extracts even rows, and of those, only the columns from -1 to 1, from `c`. These values are then copied into the first five rows of `b`. Instead of subscript triplets, one can also use an integer array as an index for another array. This is shown in line 9. Fortran provides other ways to modify or reinterpret arrays, including the `reshape` function that can change the number of dimensions and the size in each dimension, and has optional arguments to pad or reorder an array.

When an array is passed to a function or subroutine as an argument, it may be accessed with a different index scheme inside that function or subroutine. For example, a three-dimensional array with index ranges $[0:8][0:2][0:2]$ could be passed to a subroutine that internally declares this argument as an array with ranges $[1:9][1:3][1:3]$ or $[0:8][0:8]$ or any other number of dimensions or index ranges, as long as the array within the callee has at most as many overall entries as the array within the caller. This essentially provides a *view* of the original array, and because Fortran uses the call-by-reference paradigm, any changes to this re-interpreted array within the callee will also affect the original array in the caller. Depending on the situation, the Fortran compiler may implement this using an array descriptor and suitable index expressions, or by transparently copying data to and from an array that is used within the callee.

A similar situation occurs when Fortran pointers are used. Despite their similar name with C pointers, their behavior and features differ significantly. Fortran pointers can represent a view into a multi-dimensional array, and contain size and shape information. For example, a pointer can be associated with an array slice that represents column 1 across all rows in an array, as shown in line 11 of Figure 5. In this case, writing to the first element in `p` will also modify the first row in `c`'s column 1. The `size` and `shape` functions can be used on `p` and will return the size and shape of the portion of c that p is associated with. The pointer itself can be accessed with a subscript triplet or index array, and the pointer can be passed to a subroutine or function that may reinterpret it with a different dimensionality or index range.

There are a number of details regarding the use of arrays and pointers in Fortran that we do not discuss in this paper for brevity. We refer to [1] (particularly Sections 5.4, 5.6 and 12.6.4) for a more thorough discussion.

4.2 Modeling Fortran Arrays for Verification

Arrays in CIVL-C always have indices starting at 0 and do not support strides, sectioning, or reshaping. To handle the features described in the previous subsection, each Fortran array is modeled by a CIVL-C array that is augmented with a recursive data structure called FORTRAN_ARRAY_DESCRIPTOR. This allows CIVL to model the rich Fortran array semantics using only CIVL-C language features. As Figure 6 shows, the descriptor stores metadata for an array instance, and contains the kind, rank, index upper and lower bounds and strides, as well as a pointer.

When a Fortran program creates a new array from scratch, CIVL will create a CIVL-C array whose length is the total number of elements in the Fortran array. This array is then augmented with an array descriptor whose kind is SOURCE and whose pointer holds the memory address of the CIVL-C array. The bounds and stride in the descriptor are set according to those set by the Fortran program. In essence, the descriptor provides a mapping from the Fortran array index (which may be strided or non-zero-based) into the CIVL-C array index (which is dense and zero-based). This mapping is used by the CIVL-C program whenever the Fortran program accesses the array.

If a Fortran array instance is created by reshaping or sectioning an existing array, no new CIVL-C array is created. Semantically, the new array instance in Fortran provides a view into the existing array, which we model by creating a new array descriptor with appropriate bounds and stride whose kind is

```c
typedef struct FORTRAN_ARRAY_MEMORY *farr_mem;
typedef struct FORTRAN_ARRAY_DESCRIPTOR *farr_desc;
typedef enum FORTRAN_ARRAY_DESCRIPTOR_KIND {
  SOURCE, // A var. decl. w/ an array type or a dimension attr.
  SECTION, // An array section
  RESHAPE // An array, whose indices are reshaped w/ no cloning
} farr_kind;
struct FORTRAN_ARRAY_DESCRIPTOR {
  farr_kind kind; // The kind of a Fortran array descriptor
  unsigned int rank; // The rank or the number of dimensions.
  int *lbnd; // A list of index left-bounds for each dim.
  int *rbnd; // A list of index right-bounds for each dim.
  int *strd; // A list of index stride for each dim.
  farr_mem memory; // Being non-null iff kind is 'SOURCE'
  farr_desc parent; // Being non-null iff kind is NOT 'SOURCE'
};
```

Fig. 6. Implementation of the CIVL-C array descriptor.

```fortran
PROGRAM ARRAYOP
  INTEGER :: A(0:8)
  CALL SUBR(A(1:7:2))
! A: {0,1,0,2,0,3,0,4,0}
END PROGRAM ARRAYOP

SUBROUTINE SUBR(B)
  INTEGER :: B(-1:0, 2:3)
  B(-1, 2) = 1
  B(-1, 3) = 2
  B( 0, 2) = 3
  B( 0, 3) = 4
END SUBROUTINE SUBR
```

```c
int main() {
  fa_desc A = fa_create(sizeof(int), 1, {{0},{8},{1}});
  fa_desc __arg_A = fa_section(A, {{1},{7},{2}});
  subr(__arg_A);
  fa_destroy(__arg_A); ! pop section descriptor
  fa_destroy(A); ! free array descriptor and data storage
}
void subr(fa_desc __B) {
  fa_desc B = fa_reshape(__B, 2, {{-1,2},{0,3},{1,1}});
  *(int*)fa_subscript(B, {-1,2}) = 1;
    ...
  *(int*)fa_subscript(B, {0,3}) = 4;
  fa_destroy(B); ! pop reshape descriptor
}
```

Fig. 7. Transformation of array section and reshape operations

set to SECTION or RESHAPE, and whose pointer stores the location of the array descriptor for the existing array. This new descriptor now provides a mapping from indices of the new array instance into indices of the existing array instance. Such an array section or reshaped array can itself be reshaped or sectioned by the Fortran program, which will result in a stack of array descriptors. Whenever the Fortran program accesses an array at a given index, CIVL will recursively use the mappings provided by the descriptors until the index in the underlying CIVL-C array is resolved by a descriptor of kind SOURCE. Figure 7 shows how some basic Fortran array operations are translated to CIVL-C using the array descriptor and associated utility functions.

5 Evaluation

The first goal of this evaluation is to determine whether CIVL correctly verifies or finds defects in a suite of synthetic Fortran programs that use various language features peculiar to Fortran. The second goal is to investigate how CIVL performs on Fortran code from an existing production-level HPC application.

5.1 Compute Environment and Experimental Artifacts

All CIVL executions were conducted on a TACAS 2022 Artifact Evaluation Virtual Machine (AEVM) with Ubuntu 20.04; the version of CIVL is 1.21. All SMACK executions were conducted on a TACAS 2020 AEVM provided by the authors of [13]; the version of SMACK is 1.9.1. Both virtual machines were deployed by Oracle VirtualBox 6.1 on a laptop running MacOS 11.6.2 on a 2.5 GHz Quad-Core Intel Core i7 CPU with x86_64 architecture and 16 GB memory. The CIVL program and all experimental artifacts can be downloaded from https://vsl.cis.udel.edu/tacas2022.

5.2 Specification and Verification Approach

As shown in Figure 8, CIVL primitives are inserted as structured comments for verifying a Fortran code, which have no effect on the normal build process. Similar directives exist for C. These primitives have two major kinds: type qualifiers

```
PROGRAM civl_primitive_example
!$CVL $input
  INTEGER :: arg
  INTEGER :: x
!$CVL $assume(-1 .LE. arg .AND. arg .LE. 1);
  x = arg
!$CVL $assume(x .LT. 0);
    x = ABS(x)
!$CVL $assert(0 .LE. x .AND. x .LE. 1);
END PROGRAM civl_primitive_example
```

Fig. 8. Example illustrating CIVL Fortran primitives.

and verification statements. `$input` specifies that the variable in the following declaration is to be initialized with an unconstrained value of its type. The value can be subsequently constrained with an assumption statement. Alternatively, an input variable may be given an exact concrete value on the command line. Input variables are read-only.

The `$output` qualifier declares a variable to be write-only. Output variables are used for functional equivalence verification. When two programs have the same input and output variables, they can be compared to determine whether, given the same inputs, the two programs will produce the same outputs. This is carried out by CIVL's *compare* command, which merges the two programs into a single program with a new driver. The driver invokes the two programs in sequence on the same input variables, and then asserts that the corresponding outputs agree.

A CIVL assumption statement has the form `$assume(expr);`. It is used to constrain the set of executions that are considered to be valid. If an assumption is violated, no error is reported; instead, the execution is ignored and the search backtracks immediately. `$assert(expr);` reports an assertion violation if the argument expression does not hold. This statement provides the capability of checking desired properties in Fortran, which has no intrinsic assertion procedure. All primitives must be preceded by the prefix `!$CVL`.

5.3 Fortran Verification Benchmark Suites

Our suite incorporates the 22 synthetic examples from the SMACK suite [13]. These examples cover basic Fortran structures ranging from expressions to functions and subroutines. The only change made is to switch SMACK-style assertions and symbolic value assignment to CIVL primitives. SMACK uses calls of the form `assert(`*expr*`)` to check desired properties, which is similar to CIVL's `$assert` primitive. With SMACK, symbolic values are generated by calling `__verifier_nondet_int()` and assigning the result to a variable, while CIVL uses the `$input` qualifier.

To these, we added 13 examples we created ourselves, exercising different language features, including argument intent specification, array sectioning, and boolean expressions that might lead to different results if short-circuiting is or is not used. We include a parallel example that uses an OpenMP for loop, executed

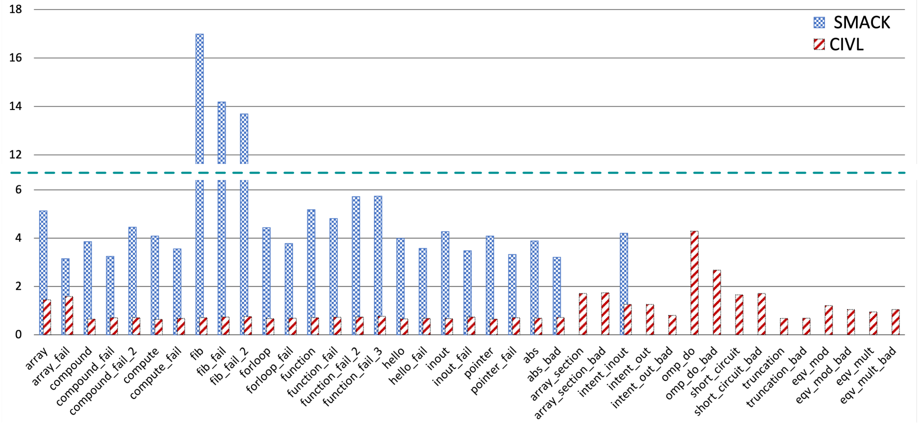


Fig. 9. Total verification time (in seconds) for CIVL and SMACK on benchmarks. Each time is the mean over 5 of 7 executions after dropping the shortest and longest.

with 4 threads. Finally, we constructed 4 pairs of programs each of which can be compared for functional equivalence.

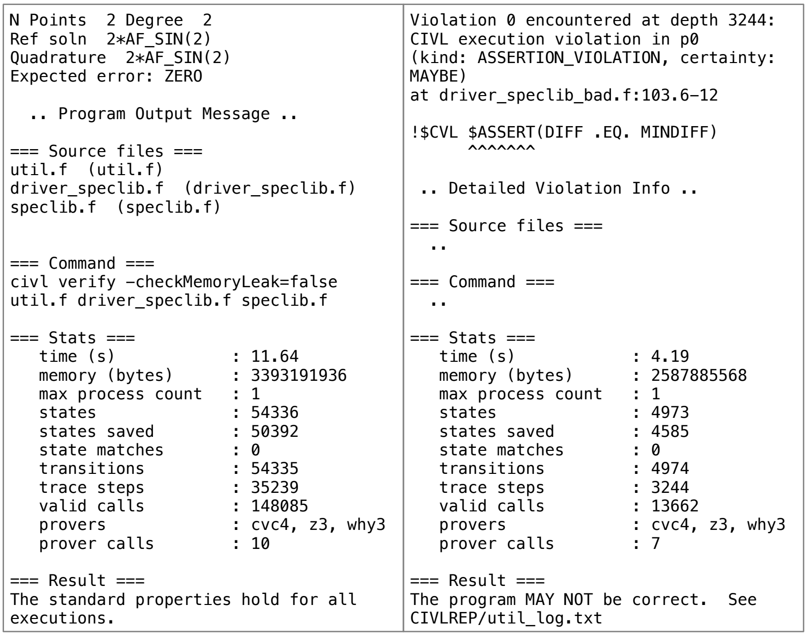
The programs are listed on the x-axis in Figure 9. Where the name includes "fail" or "bad", a negative verification result is expected; otherwise, a positive result is expected. The figure also shows the average verification execution time printed by CIVL and SMACK on each example. CIVL has correct results in all cases, while SMACK encounters exceptions or has incorrect results for some of the CIVL Fortran examples. Thus, the figure only reports timing results when the verification results are correct.

5.4 Verifying Nek5000 Components

Nek5000 [21] is a computational fluid dynamics code for simulating unsteady incompressible two- or three-dimensional fluid flow. Nek5000 has hundreds of industrial and academic users and won a Gordon Bell prize for its scalability on high performance compute clusters.

The code contains many Fortran subroutines that perform a numerical computation that can be easily expressed in a formal way. For example, there are various implementations for matrix multiplication, each optimized for best performance on a particular matrix size. We use CIVL to verify that these subroutines indeed compute matrix multiplications, by showing their equivalence with a straightforward un-optimized textbook implementation.

Furthermore, Nek5000 contains subroutines to numerically approximate the integral of a function, a process known as quadrature. Quadrature rules typically define carefully chosen locations, known as quadrature points, at which the function in question is evaluated. The results are then each multiplied with a weight, and summed to obtain the overall integral. The quality of a quadrature rule is often evaluated by quantifying its order of accuracy, where a higher order



```
N Points  2 Degree  2
Ref soln  2*AF_SIN(2)
Quadrature  2*AF_SIN(2)
Expected error: ZERO

  .. Program Output Message ..

=== Source files ===
util.f  (util.f)
driver_speclib.f  (driver_speclib.f)
speclib.f  (speclib.f)


=== Command ===
civl verify -checkMemoryLeak=false
util.f driver_speclib.f speclib.f

=== Stats ===
   time (s)            : 11.64
   memory (bytes)      : 3393191936
   max process count   : 1
   states              : 54336
   states saved        : 50392
   state matches       : 0
   transitions         : 54335
   trace steps         : 35239
   valid calls         : 148085
   provers             : cvc4, z3, why3
   prover calls        : 10

=== Result ===
The standard properties hold for all
executions.
```

```
Violation 0 encountered at depth 3244:
CIVL execution violation in p0
(kind: ASSERTION_VIOLATION, certainty:
MAYBE)
at driver_speclib_bad.f:103.6-12

!$CVL $ASSERT(DIFF .EQ. MINDIFF)
      ^^^^^^^


 .. Detailed Violation Info ..

=== Source files ===
  ..

=== Command ===
  ..

=== Stats ===
   time (s)            : 4.19
   memory (bytes)      : 2587885568
   max process count   : 1
   states              : 4973
   states saved        : 4585
   state matches       : 0
   transitions         : 4974
   trace steps         : 3244
   valid calls         : 13662
   provers             : cvc4, z3, why3
   prover calls        : 7

=== Result ===
The program MAY NOT be correct.  See
CIVLREP/util_log.txt
```

Fig. 10. CIVL output for verifying correct and erroneous Nek5000 examples

quadrature rule yields the exact result for polynomials of a higher degree. The Gauss-Lobatto Legendre quadrature rules are a unique set of weights and points that are known to be optimal under certain conditions, and are used in Nek5000. We use CIVL to verify that the quadrature implemented in Nek5000 indeed has the claimed order of accuracy, by verifying that the quadrature is exact for polynomials with symbolic coefficients of the claimed degree. Due to its uniqueness properties, this also proves that Nek5000 indeed uses Gauss-Lobatto Legendre weights and points.

We also seeded some of these implementations with defects and confirmed that CIVL reports the defects. Figure 10 shows the output from CIVL on a correct and incorrect example from Nek5000. Table 1 shows the verification results for the Nek5000 excerpts for various parameter values. The expected result is obtained in all cases, at modest cost (at most 12 seconds).

6 Related Work

Fortran has been the focus of early program verification research. One of the first papers on symbolic execution dealt with Fortran [8], and one of the earliest verification condition generation tools was for Fortran [6]. More recently, several Fortran static analyzers have been developed, including ftnchek [19], Cleanscape FORTRAN-Lint [9], and FORCHECK/Coverity [35]. These tools detect certain

Name	LoC	Result	Scale	Time	States
speclib	560	True	$2 \leq \texttt{NP} \leq 2$; $2 \leq \texttt{DEG} \leq 3$	5.14s	10857
speclib	560	True	$2 \leq \texttt{NP} \leq 3$; $2 \leq \texttt{DEG} \leq 5$	12.08s	55908
speclib_bad	560	False	$2 \leq \texttt{NP} \leq 2$; $2 \leq \texttt{DEG} \leq 3$	4.67s	6011
speclib_bad	560	False	$2 \leq \texttt{NP} \leq 3$; $2 \leq \texttt{DEG} \leq 5$	4.27s	3223
mxm_unroll	458	Eqv	3×3	5.49s	26867
mxm_unroll	458	Eqv	4×4	8.51s	59914
mxm_unroll_bad	458	NEq	3×3	5.48s	26865
mxm_unroll_bad	458	NEq	4×4	8.56s	59912
mxm_pencil	458	Eqv	2×2	5.83s	9264
mxm_pencil	458	Eqv	3×3	7.38s	26893
mxm_pencil	458	Eqv	4×4	10.14s	59968
mxm_pencil_bad	458	NEq	2×2	6.01s	9262
mxm_pencil_bad	458	NEq	3×3	7.53s	26891
mxm_pencil_bad	458	NEq	4×4	10.48s	59966

Table 1. Results of verifying Nek5000 code excerpts at various scales

pre-defined generic defects, such as variables that are read but never written, unused variables and functions, and inconsistencies in common block declarations. They do not allow one to specify and verify functional correctness properties.

Other tools use dynamic analysis (or a combination of static and dynamic analysis) to check such generic properties. One example uses the PIPS compiler to detect forbidden aliasing in subroutines [22]. The NAG Fortran compiler can also insert checking code to catch many defects at runtime [29].

In contrast, CamFort [27] implements a lightweight specification and static analysis approach. The user annotates the Fortran program with comments in a domain specific language for specifying array access patterns (stencils) or associating units of measurements to variables. CamFort, which is written in Haskel, parses the code, constructs an AST, and verifies conformance to the properties using Z3. This approach strikes a balance between the generality of program verifiers such as CIVL, which can specify arbitrary assertions in a general purpose assertion language, and the more tractable static analysis tools.

Several tools have been developed to translate Fortran to other languages. These include f2c [11] (which translates to C) and Fable [14] (C++). In addition to the issues discussed in Section 3.1, the potential of these tools as front ends for verifiers is limited by the fact that the translated code is often considerably more complex than the original or involves complex libraries which the verifier must also understand. It should be noted that Fable's approach to modeling Fortran arrays is similar to ours in that it defines a class that bundles a reference to the data with meta-data describing the "view" of the array.

A number of verification tools work off of the LLVM compiler's low-level intermediate language, LLVM IR. These include SMACK [30], Divine [2], LLBMC [34], and SeaHorn [15]. In theory, this should allow one to chain together any of the many compiler front ends that generates LLVM IR with a general LLVM

IR verifier. In practice, this is very difficult, and most of these verifiers accept only a subset of LLVM IR generated by a particular front end from a particular source language—usually C or C++ [13]. To the best of our knowledge, only SMACK has been applied to Fortran [13], using the Flang front end [12]. However, the subset of Fortran accepted and the example codes themselves are small. A more significant concern, discussed in Section 3, is that a front end may "compile away" defects in the source program by choosing one of several acceptable ways to translate a construct with unspecified behavior, or assuming the absence of undefined behaviors.

In this work we have translated Fortran to the intermediate verification language (IVL) CIVL-C. Other, more widely-used, IVLs include Boogie [3] and Why3 [5]. Among these languages, CIVL-C stands out for its robust support for pointers and concurrency, which simplifies much of the modeling effort.

The CIVL verifier analyzes a CIVL-C program using symbolic execution, a widely-used technique for test-case generation and verification. Other mature symbolic execution tools include KLEE [7] (for C programs, via LLVM) and Symbolic PathFinder [23] (for Java byte code).

7 Conclusion and Future Work

We presented a Fortran extension to CIVL, a novel model-checking approach that preserves and reveals defects in source code written in Fortran. Compared with compiler-based verifiers, this tool parses and analyzes source programs from a verification perspective. In doing so, it mitigates against the risk of missing defects that are eliminated via legal but non-defect-preserving compiler optimizations.

The extension includes a data structure and associated algorithms for describing Fortran array metadata and tracking complex array transformations. This method of handling Fortran arrays could be adopted by other verification tools. The extension also supports a set of CIVL verification primitives which can be introduced into Fortran programs as structured comments.

Evaluation results show that our tool performs correctly and quickly (compared to previous work) on a range of synthetic benchmarks and some kernels extracted from real world applications. In the future, we plan to enlarge the supported subset of Fortran language features and to enhance support for verifying Fortran programs with OpenMP directives. The resulting CIVL extension is expected to cover the DataRaceBench [36] suite, including both the C and Fortran examples.

Acknowledgements

This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, and by contracts DE-AC02-06CH11357 and DE-SC0021162. Support was also provided by U.S. National Science Foundation award CCF-1955852.

References

1. Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T.: The Fortran 2003 Handbook: the Complete Syntax, Features and Procedures. Springer Science & Business Media (2008). https://doi.org/10.1007/978-1-84628-746-6
2. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium (FMCO 2005). Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, 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), http://dl.acm.org/citation.cfm?id=2032305.2032319
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), http://proval.lri.fr/publications/boogie11final.pdf
6. Boyer, R.S., Moore, J.S.: A verification condition generator for Fortran. Tech. Rep. CSL-103, SRI International, Computer Science Laboratory, Menlo Park, CA (June 1980), https://apps.dtic.mil/sti/pdfs/ADA094609.pdf
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08) (2008)
8. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **2**, 215–222 (May 1976). https://doi.org/10.1109/TSE.1976.233817
9. Cleanscape Software International: FORTRAN-lint: a pre-compile analysis tool, https://stellar.cleanscape.net/docs_lib/data_F-lint2.pdf, accessed 13-Oct-2021
10. Dingle, N.: Not only Fortran and MPI: POP's view of HPC software in Europe, https://pop-coe.eu/blog/not-only-fortran-and-mpi-pops-view-of-hpc-software-in-europe, accessed 14-Oct-2021
11. Feldman, S.I.: A Fortran to C converter. SIGPLAN Fortran Forum **9**(2), 21–22 (Oct 1990). https://doi.org/10.1145/101363.101366
12. Flang Fortran language front-end. https://github.com/flang-compiler/flang, accessed 09-Oct-2021
13. Garzella, J.J., Baranowski, M., He, S., Rakamarić, Z.: Leveraging compiler intermediate representation for multi- and cross-language verification. In: Beyer, D., Zufferey, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 90–111. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_5
14. Grosse-Kunstleve, R.W., Terwilliger, T.C., Sauter, N.K., Adams, P.D.: Automatic Fortran to C++ conversion with FABLE. Source Code for Biology and Medicine **7**(5) (2012). https://doi.org/10.1186/1751-0473-7-5

15. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9035, pp. 447–450. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_41
16. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO'04). pp. 75–86. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
17. Lawrence Livermore National Laboratory: CORAL benchmark codes (2014), https://asc.llnl.gov/coral-benchmarks, accessed 14-Oct-2021
18. Message Passing Interface Forum: MPI: A Message-Passing Interface standard, version 3.1 (Jun 2015), https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
19. Moniot, R.K.: ftnchek: a static analyzer for Fortran 77, https://www.dsm.fordham.edu/~ftnchek/, accessed 09-Oct-2021
20. 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, 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
21. NEK5000: a fast and scalable high-order solver for computational fluid dynamics (2021), https://nek5000.mcs.anl.gov, accessed 14-Oct-2021
22. Nguyen, T.V.N., Irigoin, F.: Alias verification for Fortran code optimization. Electronic Notes in Theoretical Computer Science **65**(2), 52–66 (2002). https://doi.org/10.1016/S1571-0661(04)80396-7, COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002)
23. Noller, Y., Păsăreanu, C.S., Fromherz, A., Le, X.B.D., Visser, W.: Symbolic Pathfinder for SV-COMP. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 239–243. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_21
24. NVIDIA: CUDA Toolkit Documentation, v11.4.2, https://docs.nvidia.com/cuda/, accessed 14-Oct-2021
25. Open Group: IEEE Std 1003.1: Standard for information technology—Portable Operating System Interface (POSIX(R)) base specifications, issue 7: pthread.h (2018), https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html
26. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2020), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, version 5.1
27. Orchard, D., Contrastin, M., Danish, M., Rice, A.: Verifying spatial properties of array computations. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–30 (Oct 2017). https://doi.org/10.1145/3133899, article no. 75
28. Parr, T.: The Definitive ANTLR4 Reference. The Pragmatic Bookshelf, Dallas, TX (2013), https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/
29. Polyhedron Solutions: Linux Fortran compiler diagnostic comparisons, https://www.fortran.uk/fortran-compiler-comparisons/intellinux-fortran-compiler-diagnostic-capabilities/, accessed 13-Oct-2021

30. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementation. In: Biere, A., Bloem, R. (eds.) Proceedings of the 26th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7
31. Rasmussen, C.E., et al.: OFP: Open Fortran Project, https://sourceforge.net/p/fortran-parser/wiki/Home/, accessed 14-Oct-2021
32. Rosner, R., Calder, A., Dursi, L., Fryxell, B., Lamb, D., Niemeyer, J., Olson, K., Ricker, P., Timmes, F., Truran, J., Tufo, H., Young, Y.N., Zingale, M., Lusk, E., Stevens, R.: Flash code: Studying astrophysical thermonuclear flashes. Computing in Science and Engineering **2**, 33–41 (2000). https://doi.org/10.1109/5992.825747
33. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The Concurrency Intermediate Verification Language. In: SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, New York (Nov 2015). https://doi.org/10.1145/2807591.2807635, article no. 61, pages 1–12
34. Sinz, C., Merz, F., Falke, S.: LLBMC: A bounded model checker for LLVM's intermediate representation. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7214, pp. 542–544. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_44
35. Synopsys: Synopsys static analysis (Coverity) Fortran syntax analysis, https://community.synopsys.com/s/article/Synopsys-Static-Analysis-Coverity-Fortran-Syntax-Analysis, accessed 13-Oct-2021
36. Verma, G., Shi, Y., Liao, C., Chapman, B., Yan, Y.: Enhancing DataRaceBench for evaluating data race detection tools. In: 2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness). pp. 20–30. IEEE (2020). https://doi.org/10.1109/Correctness51934.2020.00008

NORMA: a tool for the analysis of Relay-based Railway Interlocking Systems

Arturo Amendola[1], Anna Becchi[2](✉), Roberto Cavada[2],
Alessandro Cimatti[2], Andrea Ferrando[2], Lorenzo Pilati[2],
Giuseppe Scaglione[3], Alberto Tacchella[2], and Marco Zamboni[2]

[1] Consultant for RFI System Development – Roma, Italy
amendola.arturo@yahoo.com
[2] Fondazione Bruno Kessler – Povo, Trento, Italy
{abecchi,cavada,cimatti,aferrando,lpilati,atacchella,mazamboni}@fbk.eu
[3] RFI Rete Ferroviaria Italiana – Roma, Italy g.scaglione@rfi.it

Abstract. We present Norma, a tool for the modeling and analysis of Relay-based Railways Interlocking Systems (RRIS). Norma is the result of a research project funded by the Italian Railway Network, to support the reverse engineering and migration to computer-based technology of legacy RRIS. The frontend fully supports the graphical modeling of Italian RRIS, with a palette of over two hundred basic components, stubs to abstract RRIS subcircuits, and requirements in terms of formal properties. The internal component based representation is translated into highly optimized Timed nuXmv models, and supports various syntactic and semantic checks based on formal verification, simulation and test case generation. Norma is experimentally evaluated, demonstrating the practical support for the modelers, and the effectiveness of the underlying optimizations.

Keywords: Relay-based Railway Interlocking Systems · graphical modeling · model checking

1 Introduction

Railway interlocking systems (RIS) are complex signaling apparatus that prevent conflicting movements of trains through an arrangement of tracks, most notably stations. The basic requirement is that a signal to proceed is not displayed unless the route to be used is proven safe. This means positioning the switches in the appropriate position, controlling the level crossings, and setting the aspects of the signals to indicate the expected speed restrictions.

Although the world is slowly migrating to computer-based RIS, the predominant solutions are still based on electromechanical technology, where the logic of the interlocking procedures is encoded in the evolution of the status of the circuit *relays*. RRIS are a costly and hard to modify technology. Yet, RRIS have been working correctly and safely for decades. In the migration from relay-based to computer-based RIS, it would be a natural choice to use RRIS as golden requirements for the new implementations. However, they are de facto *legacy systems*,

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 125–142, 2022.
https://doi.org/10.1007/978-3-030-99524-9_7

whose behavior is known to a handful of highly specialized domain experts, hence expensive to maintain and update. Even more, the RRIS schematics are often available only in printed form, and their behaviour needs to be manually simulated. Understanding RRIS, modeling them in digital format and extracting requirements from them is hence a major challenge in the migration process.

In this paper we present Norma, a real-world tool for the formal modeling and analysis of RRIS. Norma is the result of a research project funded by the Italian railway network company (Rete Ferrovie Italiane - RFI), within a process of reverse engineering and migration to computer-based technology of the legacy RRIS currently in operation [6]. Norma leverages formal verification techniques to provide extensive support for modeling, debugging, understanding, traceability and verification. This also enables simulation, testing, and properties extraction from RRIS.

The Norma frontend fully supports the graphical modeling of the Italian RRIS, with a palette of over two hundred types of configurable components, both on direct and alternate current, and single- and double-wired convention; it allows the use of stubs to abstract RRIS subcircuits, and to specify requirements in terms of formal properties.

Given that RRIS often contain thousands of component instances, the task of manually modeling RRIS is repetitive and error prone. To support the modeler, Norma supports a modeling style where components are accurate at the electrical level, so that the digital model is in a one-to-one correspondence with the printed schematic, and no manual abstraction is needed. Furthermore, a number of syntactic and semantic checks ease the debugging of the models. In fact, while RRIS are operating correctly, errors may be introduced in the modeling process.

RRIS graphical models are internally represented with suitable data structures, and automatically formalized in nuXmv as symbolic timed transition systems over Boolean and real-valued variables. Then, a rewriting pipeline implements several domain-specific simplification steps that take into account the features of the RRIS to produce a drastically reduced model. Notable simplifications in the pipeline include the identification and inlining of functionally dependent variables, and contextual determinization of unconstrained signals.

The simplified model is then amenable for simulation, checking of invariant and temporal properties, and test case generation, in addition to providing a number of semantic checks deriving from built-in properties.

Norma is based on an extensible software architecture. It is built on the DIA toolset, and follows a library-based approach, where each component is modeled and tested in isolation. In turn, the component library is built by means of an automated process relying on configuration tables. At the core, the verification process is carried out by the nuXmv model checker.

Norma is actively being used within RFI. We experimentally evaluated its capabilities on real-world RRIS schematics with thousands of variables, with several important findings. First, it is very effective in supporting the modelers: the semantic checks proved to be invaluable to pinpoint several subtle modeling errors. Second, the underlying optimizations dramatically reduce the computation time of the verification tasks. Third, Norma supports the automated extraction of specifications, such as the table of mutually incompatible routes encoded in the RRIS of a medium-sized station.

To the best of our knowledge, Norma is the only tool supporting the modeling and the formal analysis of real-world RRIS. It integrates behind a graphical front-end some powerful reasoning capabilities, without exposing domain experts to the intricacies of formal verification. Approaches to the formal analysis of RRIS have been proposed [5,12,11,15]. However, the RRIS is modeled at a high level of abstraction, so that important features are lost. More importantly, the user is in charge of ensuring the correspondence between the circuit schematics and the formal model. Given the typical size of real-world RRIS, the process appears to be very prone to errors. In contrast, we rely on a comprehensive approach to component-based modeling [8], where the RRIS is described as a multi-domain switched Kirchhoff network, hence supporting a precise and electrically-accurate semantics.

This paper is structured as follows. In Section 2 we present the background domain of RRIS. In Section 3 we overview the functions of Norma. Then, in Section 4, 5 and 6 we discuss in detail the front end, the compiler and the simplifier. In Section 7 we overview the software architecture of Norma, and in Section 8 we present the experimental results. In Section 9 we draw some conclusions and discuss the future developments.

2 Relay-based Railway Interlocking Systems

At the beginning of the 20^{th} century, the rapid growth in the development of railways systems called for technological solutions to avoid collisions among trains and other safety critical issues. Signals were originally installed at fixed track side positions, featuring mechanical arms which were manually operated through levers, pulleys and wires from local signal boxes. As purely mechanical devices proved soon to be very unreliable, they were substituted by electric and electromechanical devices, like for example signals with colored lights and railroad motor switches. Aside from being much more reliable and economically sustainable, these new devices could now be controlled remotely in a centralised fashion. The control procedures went from manually operating each device individually, to *logics* able to operate automatically multiple devices at once, for example to safely create and monitor an itinerary for a train to leave a station. These centralized logics were mainly based on *relays* and proved to be able to operate reliably for decades.

A relay is an electromechanical element, generally composed by a coil and one or more contacts: when the coil is traversed by sufficient current, it generates a magnetic field that will close or open the contacts depending on the relay type. When the current flow is interrupted, the contacts will return to their initial state.

By combining relays in circuits it is possible to implement a sequential logic, where the combinational part is made of series/parallel circuits of relay contacts,

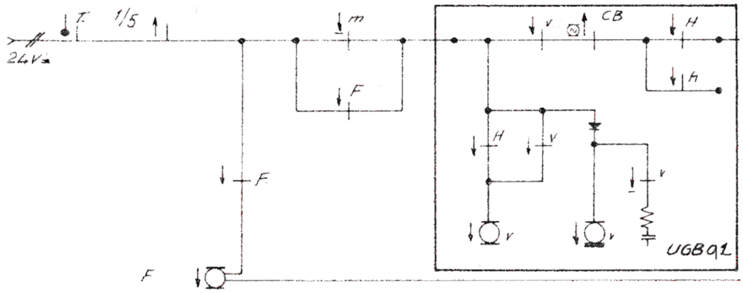


Fig. 1. Extract of schematics of itinerary from Italian legacy RIS relay logic

and the memory part is encoded in the relay coil state, i.e. being powered or not. Inputs from the environment of such logic are electrical signals coming from the rail track (e.g. train pedals), or from the user-interface (e.g. buttons or levers). Outputs to the environment are electrical commands to the plant (e.g. power to a rail crossing motor) or to the user-interface (e.g. light bulbs that represent a signal status).

The general concept of relay circuit logic is specialized in the solutions adopted in the Italian railway network. In such a domain, circuits are represented as schematics in separate sheets, along with informative material like topological schematics of the track and devices controlled by the relay logic, tables, textual notes, etc. Circuits are made of interconnected components. Components have terminals, and terminals are connected by lines representing electrical connections. Some components have an associated name to represent the relation between a coil and its contacts.

The domain has several interesting characteristics. The first one is the complexity of the domain: there is a large number of components types that differ on the timing required to operate, the amount of memory elements that can be stored, and so on. In particular, base components like coils, contacts, levers, loads, etc. can be combined with zero, one or more specifiers to specialize the components behaviour. This combination leads to more than 5000 components types which can be instantiated in a circuit. As an example, there exist dozen types of relay which are characterized by being delayed or not (when activating, deactivating, or both), polarized or not, single or double coil, stabilized or not, etc.

Second, the circuits can be operating either with direct or alternate current, where discrete signals (e.g. the maximum allowed train speed in a track segment) are encoded by means of frequency and/or amplitude modulation. Some component types can generate modulated current, and some corresponding component types can read it and react accordingly.

Third, several design conventions were adopted for the sake of readability of the relay logic circuits. Three significant cases are: (1) the logical representation of components in circuits, (2) single/double-wired circuits and (3) units. Circuits are logically represented (1) in schematics, in contrast with the physical representation found in conventional electrical schematics. In such logical representation, relay coils and their contacts are represented in separate circuits, dislocated according to logical criteria, as the coil of a single relay and its contacts may indeed belong to different logical functions. Like in computer programs, where messages can be sent and received from different logical units, coils can be activated (message sent) and contacts react accordingly (message received) in separate logical units. Separation of coils and circuits in the schematics helps preserving logical cleanness and may improve the readability of the schematics. The relation between a coil and its contacts is pragmatically kept by using the same name. See for example Fig. 1, where the coil of relay "F" (bottom left) and some of its contacts appear in the same circuit.

In single-wired circuits (2), an electrical connection line between two components implicitly represents a pair of wires, i.e. the current flows in one direction through one wire in the implicit pair, and returns through the second wire in the pair. The single-wire representation is frequently used as it is practical and readable. However, sometimes it is needed to represent explicitly the two wires, e.g. when a contact needs to cut explicitly the return wire. Sometimes the single/double-wired representation is mixed in the same schematics or even in the same circuit.

Units (3) are a pragmatic way of reusing parts of a schematics. A unit represents a generic functionality, e.g. the logic which can control a single rail switch. A unit is a set of circuits associated with a name. Other circuits can refer (indeed so instantiating the unit) to a set of components which a unit contains, by using the same component names along with the unit's name as namespace. For example, in Fig. 1, all contacts "H" belong to the unit "UGB92".

As a final remark, consider that the schematics of relay circuits are legacy, available in terms of large printouts. They have been designed along many decades, with new features added incrementally and often monotonically. This makes the logic of a medium sized station interlocking very large, containing thousands of components spread over dozens of A0 sheets.

3 Norma: overview

Norma is a tool to model, trace, understand and analyze RRIS used in the Italian railway network. The ultimate goal of Norma is to support the understanding and the reverse engineering of RRIS. This demands that all original schematics in printouts get correctly digitalized into formal models that are amenable for automated verification, e.g. by a model checker.

In the workflow supported by Norma (Fig. 2), there are three main working lanes: Modeling, Traceability and Analysis.

Modeling supports the graphical digitalization of formal models. *Traceability* keeps the links between the created models and the corresponding requirements and regulatory documents. *Analysis* supports the verification of properties about

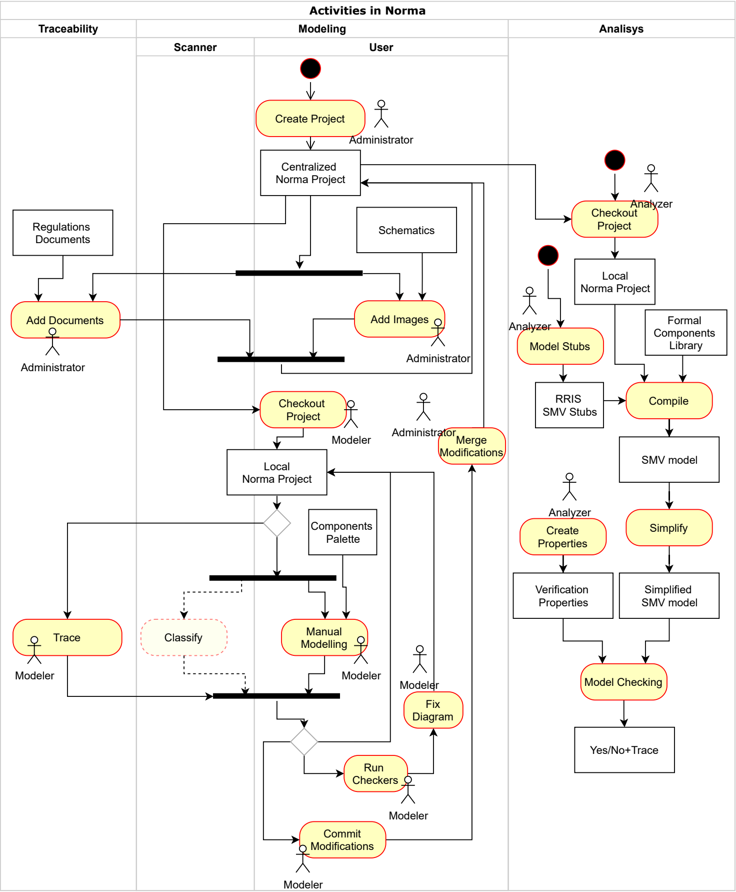


Fig. 2. High-level workflow with Norma

the RIS. These lanes are described further in Section 4 (Graphical Modeling) and Sections 5 and 6 (Analysis).

The activities involved in these lanes are performed by different users operating as a structured team: the *administrator* inserts input artifacts to enable *modelers* and *analyzers* to work on them. Each modeler works on schematics areas exclusively, and their contributions are merged by the *administrator* into the project. This avoids any risk of conflicts.

Modeling and Traceability Modeling is enabled by an *administrator* that creates a project, adds images of RRIS schematics to be modeled, adds regulatory documents for traceability, and commits the modifications to a centralized repository which all enabled users can access. *Modelers* can then checkout the project locally and graphically model components (picked from a palette), connections, units and all other parts that are relevant for the formal analysis. As the size and number of schematics is very relevant, Norma enables in its architecture the integration of an image classifier/recognizer to (semi)automatically recognize components and connections among them. This feature is currently under evaluation and not yet deployed. The *modeler* can also select regions of the modeled RRIS and associate those regions with parts in regulatory documents describing the requirements that the selected RRIS covers. The *modeler* can check the model against a set of syntactic rules (e.g. there are no floating connections) and iterate, then they can commit the local modifications to the remote repository for the *administrator* evaluation and admittance of the contribution.

Analysis The formal core of Norma is based on a compiler that transforms a graphical model into a formal model in SMV language, that can be processed by nuXmv model checker. The compiler picks components from a SMV library of timed automata with real-valued variables, each corresponding to a component type in the graphical model, and composes the networks accordingly to the electrical and logical connections among them. *Stubs* are SMV modules that *Analyzer* exploits to model abstracted parts of the RRIS which do not appear in the graphical model. The compiler injects and connects the stubs directly in the generated SMV model. The result of the compilation is a model whose size is directly related to the size of the modeled RRIS. In order to ease the formal verification process, the SMV model passes through a conservative simplification process that can dramatically reduce its size. Model checking is finally carried out by expressing LTL or invariant properties and using the nuXmv model checker.

4 Graphical modeling of RRIS

A Norma project is defined as a set of Documents and RRIS, with meta information to link them. The key idea is to allow modelers to draw the digital schematic on top of the original RRIS image. Hence, the RRIS within a project are structured in *layers* , where contents in higher levels hide the content in lower ones. Layering enables a clear separation between different types of elements, and supports the modelers during the digitalization process.

There are 5 layers: *Original RRIS*, holding the image of the original RRIS schematics; *Masks*, used to hide sections of *Original* as soon as they get modeled; *Modeled Components*, the main working layer where modelers put components and connections; *Units layer*, holding named polygons modeling Units; *Traceability*, holding several types of tracing information. By hiding/showing a layer, the modeler is able to focus on specific parts of the RRIS, e.g. to identify those parts that still need to be modeled.

Modeling mainly consists in placing components, connections and units in the diagram. The components palette (Fig. 3) has a central role in this process. As there exist such a large number of components, a customized interactive palette was designed to substitute the default, flat palette which could not be effectively used. The modeler is guided through the process of picking and configuring the

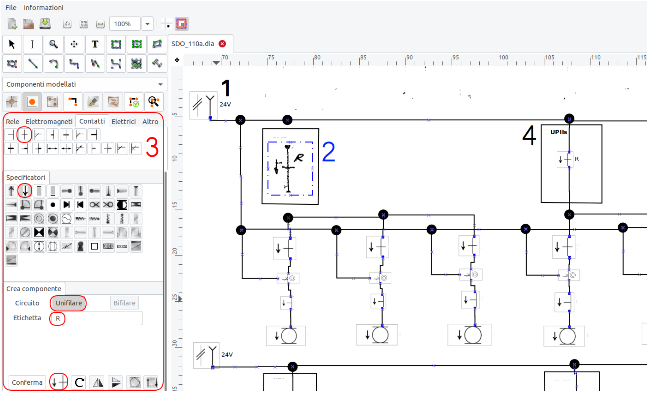


Fig. 3. 1. Modeled component 2. Not yet modeled component 3. Modeling through interactive palette 4. Example of unit

required component and specifiers matching the RRIS section being modeled. The selection begins from the *typology* (coils, contacts, load, etc.) and continues in interactive steps to allow the characterization of specifiers, single/double wiring, number of terminals, parameters, flipping and rotations, etc. The effectiveness of the process is increased by means of domain-specific constraints to restrict the user choices. These constraints are automatically generated, as described in section 7. Connections and junctions are also customized with respect to the default modeled style.

Traceability aims at linking requirements, found in regulatory documents, to the fragments of RRIS implementing them. NORMA allows the modelers to select texts and images in PDF documents, and to select regions of the model. Each selection is given automatically an ID, and IDs can be linked at project level to keep traceability information.

Utilities are made available to help the modeler. It is possible to search through traceability data and models, for example to search components by name or by type, or to search the coil corresponding to a given contact. Syntactic checkers can be run to spot errors or other issues like for example missing or wrong parameters, missing components (e.g. a contact without a corresponding coil), missing connections, etc. Selecting an issue moves the focus to the specific location in the model.

5 Compilation in Timed SMV

A RRIS is stored internally as a set of bipartite graphs $\{\mathcal{G}_i(T, N, W)\}_{i\leq n}$ where each $\mathcal{G}_i$ represents a circuit, i.e., a set of components terminals (T) connected to junctures – or nodes – (N) by wired edges (W).

The compilation converts such a description into an symbolic, infinite-state timed automaton specified using (timed) SMV, the language of the nuXmv model checker. The hierarchical structure of SMV modules enables us to produce specifications that directly reproduce the structure of the starting schematics; combined with a library of component models, we fully leverage the advantages of the compositional approach of Multi-Domain Switched Kirchhoff Networks (MDSKN) described in [8].

SMV library of components. The SMV library of components consists of 41 different formal models. Most of them depend on one or more parameters, which are automatically instantiated at network generation time, based on the parameter choices in the RRIS. The values of such parameters are either supplied directly by the user or automatically selected according to the component role in the network.

In some cases, different electrical components are mapped to the same SMV model; this happens when the differences between the components are not relevant for their electrical modeling (e.g. in case of manufacturing differences between relays).

Single-wired schemes are translated into equivalent double-wired schemes by a preliminary pass. After this, only double-wired components are considered, so that no connection is left implicit in the resulting formal model.

The interface between components is realized by means of a special *terminal* module. This module defines a pair of electrical variables ii and vv representing, respectively, the current and the voltage at the terminal, corresponding to flow and effort in the MDSKN framework [8].

In addition to electrical connections, there are *logical* connections between components, e.g. between a relay R and its contacts. To handle this kind of connections, the models in the SMV library are divided in two classes: *master* and *slave* components.

The SMV model for a master component exports the appropriate state variables (e.g. the activation status of a relay) that trigger a corresponding action in its slaves. The SMV model for a slave component is characterized by the presence of one or more *parameters*, which play the role of external inputs for the component. The connection between these inputs and the correct master outputs is then resolved when the models are composed together to form the final network, as explained in the next subsection.

Compared to the approach described in [8], that relied on a network of *hybrid automata*, supporting arbitrary continuous dynamics, here we consider networks of *timed automata* (extended with real variables), whose only continuous evolution is based on the standard clocks of the form $\dot{c} = 1$. In fact, the domain experts pointed out that a precise modeling of transient states (e.g. in RC circuits implementing an activation delay for a relay) is not necessary for the correct description of the RRIS and could be safely approximated with timing constraints. Hence, we adopt a modeling style where the continuous dynamics are replaced by a set of discrete transitions happening within a constrained time interval.

While being sufficient for all practical purposes, it supports a more adequate synchronous composition and makes the verification task easier.

Circuits composition. The SMV model $\mathcal{C}$ for a circuit $\mathcal{G}(T, N, W)$ is obtained as the synchronous composition of the models for its components, with additional constraints representing both wired and logical connections.

For each wired juncture in N, connected to terminals $t_1, \ldots, t_k$, we add to the invariant of the circuit the Kirchhoff conservation of current law $t_1.ii + \cdots + t_k.ii = 0$, and the equality of potentials law $t_i.vv = \cdots = t_k.vv$. Since all the components expose the same interface of current and potential variables at the terminals, this composition step is component-agnostic and localized to a single circuit.

Logical connections, instead, require to resolve the correct binding between master outputs and slave inputs which corresponds to the configuration of the graphical specifiers used. Then, the master output is passed as input parameter to the slave component's module.

The high level topology of the network is defined by a graph $\mathcal{N} = (\{\mathcal{C}_i\}, R)$ showing the logical connections between circuits. Namely, an oriented edge $(\mathcal{C}_i, \mathcal{C}_j)$ belongs to R iff there exists a component in circuit $\mathcal{C}_i$ which is the master of a component in circuit $\mathcal{C}_j$. The network topology may have cycles: a master component may be associated to a slave in the same circuit, therefore inducing a self loop in the graph $\mathcal{N}$. In order to preserve the causality of the events, it is important to model the remote action of a master on its input with a *transition* in the SMV model. More specifically, in every master we should use an urgent transition relation which delays the master output signal to the next state, in which it will be actually read by the slave. The analysis of $\mathcal{N}$ allows for an optimization of the SMV module for the network, which aims to shorten the paths of the resulting transition system. Namely, we insert the minimum number of delayed masters needed to break cycles.

Stubs and Assumptions. The RRIS network may have dangling inputs in the switches which respond to the status of a lever or a button controlled by a human operator. In order to have a self contained closed system, we explicitly model the environment module $\mathcal{E}$ which includes the models of the external masters.

In order to support a localized analysis of the RRIS, focusing only on a subset of the circuits in the network and abstracting the others, NORMA supports the addition of a *stub* module $\mathcal{S}$, which includes the masters belonging to removed circuits.

Both the environment and the stub modules can be used to model assumptions on the language of the inputs read by the network. As an example, when modeling the inputs coming from the trackside, we can add nominal or faulty assumptions on the behaviour of the signals. Such hypotheses can be provided by the user directly as SMV constraints. Furthermore, in order to ease the task for railway experts (e.g. having to define sophisticated or repetitive assumptions), NORMA also supports the automatic translation of such constraints from standardized Excel spreadsheets.

6 Simplification of RRIS models

NORMA contains a simplifier to reduce the number of variables, especially the real-valued ones, in the SMV models of the RRIS produced by the compiler. The steps of the simplifier are conservative: the optimized model is equivalent to the previous one with respect to the observable variables, i.e., the ones which can be included in a property to verify.

6.1 Equivalence propagation

Due to the compositional approach, the variables involved in the invariants of the circuits are highly redundant. Current and voltage variables are exposed by all component terminals and are strongly interconnected by Kirchhoff laws. For this reason, the first simplification step tries to reduce the number of variables by inlining equivalences. Namely, we clusterize the real variables into equivalence classes, propagating the equivalences that can be inferred syntactically from atoms of the form $x = y$ or $x + y = 0$. Variables are substituted with a unique representative element for their equivalence class, and clusterization is repeated until fixpoint.

6.2 Abstracting electrical variables

The variables occurring in the invariant of each module of the network (circuits, environment or stub) can be classified exploiting the information about the topology as follows.

- Input variables I: boolean variables possibly defined in other circuits used to model the open / closed condition of a switch. Each configuration of these variables correspond to a *discrete mode* of the circuit.
- State variables including: real-valued clock variables C, used to model timers inside components; history boolean variables H, needed in some component to keep track of the previous state; all the real-valued electrical variables E used for the values of current and voltage in each terminal.
- Output variables including: boolean variables representing the exposed master outputs Q, such as the status of a coil; real-valued *probes* P. Probes are added to the circuit to pin the points in which one would like to read the values of current or voltages.

We simplify the model by removing the electrical variables which are only needed to establish a binding from the switches to the relays and probes. As a matter of fact, the input and output variables are the only observable ones, i.e., they can be used in the specifications to verify. This simplification step is based on the fact that electrical variables do not evolve during timed transitions.

Pseudocode 1 Removal of electrical variables from the network

```
function REMOVE-ELECTRICAL-VARS($\mathcal{N} = (\{\mathcal{C}_i\}, R)$)
    for all $\mathcal{C}_i$ do
        classify vars in $\mathrm{Invar}_i(I, C, H, E, Q, P)$
        $\mathrm{Invar}_i$ := DETERMINIZE-VARS($\mathrm{Invar}_i$);
        $\mathrm{Invar}_i$ := QUANTIFY-ELECTRICAL-VARS($\mathrm{Invar}_i$);

function DETERMINIZE-VARS($\mathrm{Invar}(I, C, H, E, Q, P)$)
    if sat $(\mathrm{Invar} \wedge \mathrm{Invar}[Q \,/\, Q'] \wedge (Q \neq Q'))$ then        ▷ Check boolean outputs
        throw error: $Q$ variables are non-deterministic.
    $\psi := \mathrm{Invar}_i \wedge \mathrm{Invar}_i[E \,/\, E', P \,/\, P']$                        ▷ Check real vars
    for all $x \in (E \cup P)$ do
        if sat $(\psi \wedge (x \neq x'))$ then
            throw warning: variable $x$ is non-deterministic
            if $x \in P$ then
                Invar := Invar $\wedge$ GET-DEFAULT-CON($x$)
    return Invar

function QUANTIFY-ELECTRICAL-VARS(Invar)
    $\mathrm{Def_C} := \{ b_{\alpha(c)} \leftrightarrow \alpha(c) \mid c \in C, \alpha(c) \in \mathrm{atoms}(\mathrm{Invar}) \}$
    $\mathrm{Def_P} := \{ b_{(p=v)} \leftrightarrow (p = v) \mid p \in P, v \in$ GET-VALUES$(\mathrm{Invar}, p) \}$
    $\mathrm{Invar}' := \mathrm{Invar} \wedge \mathrm{Def_C} \wedge \mathrm{Def_P}$
    $\phi := \mathrm{qelim}\,(\exists E, C, P \,.\, \mathrm{Invar}')$
    $\mathrm{Invar}(I, C, H, Q, P) := \phi[b_{(p=v)} \,/\, (p = v), \ldots, b_{\alpha(c)} \,/\, \alpha(c), \ldots]$
    return Invar
```

In fact, the continuous evolution of currents and voltages (e.g., the exponential dynamic of the charging process of a capacitor) is pragmatically abstracted in the modeling of the components using clock variables, which define lower and upper time bounds connecting two stationary conditions (e.g. no vs full charge). It follows that the electrical variables should be uniquely determined by the discrete modes, and that the outputs (Q and P) are function of only the inputs I, clocks C and history variables H.

Function REMOVE-ELECTRICAL-VARS reported in Pseudocode 1, firstly checks and solves the non-determinism of the outputs, then quantifies out the electrical variables. Observe that circuits can be analyzed independently and in parallel.

Solving non-determinism. In a purely theoretical setting, Kirchhoff laws may actually allow unconstrained electrical variables. For example, current can separate non-deterministically in a branching node if the wires of a loop have no load; similarly, the voltage of a terminal which is neither connected neither to a ground nor to a power source is non-deterministic. Non-determinism of real variables is an issue only if variables in P are affected, e.g. in case we verify that an unpowered terminal has always a null value. Nonetheless, reporting *all* the non-deterministic real-valued variables (including the ones in E) is a very important checker for the validation of the model: if many non-deterministic

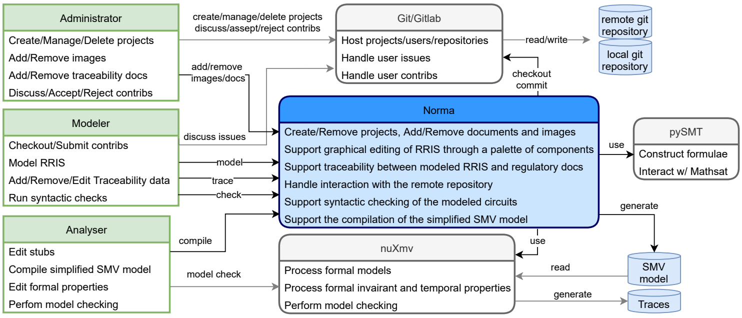

Administrator
Create/Manage/Delete projects
Add/Remove images
Add/Remove traceability docs
Discuss/Accept/Reject contribs
create/manage/delete projects
discuss/assept/reject contribs
add/remove
images/docs
Git/Gitlab
Host projects/users/repositories
Handle user issues
Handle user contribs
read/write
remote git repository
local git repository
checkout
commit
Modeler
Checkout/Submit contribs
Model RRIS
Add/Remove/Edit Traceability data
Run syntactic checks
discuss issues
model
trace
check
Norma
Create/Remove projects, Add/Remove documents and images
Support graphical editing of RRIS through a palette of components
Support traceability between modeled RRIS and regulatory docs
Handle interaction with the remote repository
Support syntactic checking of the modeled circuits
Support the compilation of the simplified SMV model
use
pySMT
Construct formulae
Interact w/ Mathsat
generate
Analyser
Edit stubs
Compile simplified SMV model
Edit formal properties
Perfom model checking
compile
model check
use
nuXmv
Process formal models
Process formal invairant and temporal properties
Perform model checking
read
SMV model
generate
Traces

Fig. 4. System level architecture of NORMA

variables are found in a circuit, then it is possible that a connection has been missed during the manual modeling.

Function DETERMINIZE-VARS checks that the boolean outputs are uniquely defined and reports to the user the set of non-deterministic real-valued variables. For non-deterministic probes, it also enriches the invariant assigning a default unique value (e.g., the null value for potentials) for the configurations of inputs in which they are under-specified.

Removing electrical variables. We want to compute $\text{Invar}'(I, C, H, Q, P) \doteq \exists E\ .\ \text{Invar}$, where the real-valued variables C and P are preserved. In order to avoid a possibly expensive geometric projection [14], function QUANTIFY-ELECTRICAL-VARS initially builds a boolean encoding for them, that enables the use of more efficient All-SMT quantifier elimination [13].

For clock variables we add a boolean variable $b_{\alpha(c)}$ for every linear constraint $\alpha(c) \in \text{atoms}(\text{Invar})$ involving a $c \in C$.

Probe variables, instead, occur in atoms mixed with other electrical variables that are to be quantified. Thanks to the previous determinization step, we know that each variable $p \in P$ can assume a finite number of values, induced by the finite configurations of the inputs. Therefore, a boolean encoding for p is obtained by enumerating the values that it can have in Invar with function GET-VALUES.

After the projection is performed on a purely boolean target, the original C and P are recovered by substituting the boolean hooks with the corresponding atoms.

7 Software architecture

At system level NORMA interacts with several entities (Fig. 4). NORMA is built on top of DIA [2], a mature open source program for drawing diagrams. To handle the interactions with the remote repository, NORMA uses GIT [1] and GITLAB [4]. To perform the SMV model simplification task NORMA uses NUXMV [7] and PYSMT [10]. The traceability block is implemented by using POPPLER [3] as backend for visualizing and annotating PDF documents. The domain-specific palette is automatically generated out of a set of tables filled in by the domain experts . These tables contain logical constraints over base components and specifiers, such as their graphical aspects, compatibility matrices, physical properties, admissible terminal configurations, orientation constraints, etc. Base components and specifiers are combined into a generated palette of over five thousand component types that can be directly read by Dia.

Norma is the result of about 2 years development, with a team of 2 to 4 person/year. Extensions made on top of Dia are implemented in C and Python, for a total of about 20KLOC. The compiler is implemented in Python and counts about 10KLOC. Norma is developed with an agile process, adopting continuous integration, feedback and testing from domain experts, and formal verification of the model library. Norma is a proprietary software and is currently not licensed to third parties.

8 Experimental Evaluation

Norma has been used for several months by a team of 3 domain experts to model several schematics of the Italian railway logic. For this experimental evaluation, we consider two reference schemas: `r-switch` and `routes48`. `r-switch` is a complete RRIS controlling a railway switch; it represents a general network which is replicated and connected to other schemas. `routes48` describes the route formation for a medium sized interlocking station, with 48 shunting routes. Each route is associated to a button and can be enabled/disabled by a human operator.

Modeling support. The considered schemas include thousands of interconnected components and several units. The modeling activity in Norma took approximately 1 person/week for each RRIS. Several further iterations were required. The checkers implemented within the graphical interface were able to immediately report to the user syntactic errors, like dangling terminals or misspelled names. Subsequent analyses, like the classification of the circuits variables and the checks of deterministic outputs, pinpointed several errors that had been missed by the syntactic checks: swap of identifiers between components (resulting in wrong logical connections between master and slaves), missing connections in n-ary junctures, and wrong initial condition for switches. The fixes were validated by examining simulations and by verifying basic properties on the corresponding SMV model.

Stubs and assumptions also proved very useful in modeling. RRIS `routes48` is connected to 11 railway switches, which have been excluded from the modeling and abstracted by a stub. Assumptions on their behaviour were expressed in tabular format, and automatically imported in Norma by way of a dedicated conversion module. Assumptions were also used to specify typical scenarios constraining the actions of the human operator. RRIS `r-switch` is attached to a stub which abstracts the behaviour of some physical entities in both nominal and faulty modalities.

Table 1. Effects of the simplifications on the number of real-valued variables and times (in seconds) spent in each phase.

						simplifier				
	compiler					load	inliner		E removal	
RRIS	#comps	#circuits	#bools	#reals	time	time	#reals	time	#reals	time
`r-switch`	185	16	94	1362	2	7	325	6	6	26
`routes48`	691	13	580	8397	14	205	1467	188	1170	21
`routes02`	51	7	40	642	2	2	108	2	5	1
`routes04`	95	8	74	1220	3	5	214	5	8	4
`routes06`	149	9	102	1935	3	12	337	11	124	12
`routes12`	210	9	139	2629	3	20	473	19	347	3

Effectiveness of Simplifications. In Table 1 we evaluate the impact of the simplification steps on the analyzed RRIS, including a set of handcrafted scaled versions of `routes48`, which control a reduced number of routes.

Column "compiler" shows the features of the obtained SMV models in terms of the number of components, circuits, boolean and real-valued variables, together with the time spent for the compilation. Column "load" reports the time spent for the untiming (with timed nuXmv [9]) and the conversion to pySMT formulae. Column "inliner" shows the effects of the propagation of equivalences, which drastically reduces the number of real variables. Column "E removal" corresponds to both the determinization and the quantification of the real variables. While the determinization check is always performed, in these experiments we heuristically enable the removal of electrical variables circuit-wise, depending on the number of input variables. As a result, the un-needed electrical variables were fully removed only in the smaller circuits. In RRIS such as `r-switch`, `routes02` and `routes04`, we obtained a simplified model where the left real-valued variables are only clocks and probes. Finally, observe that the reported time for this simplification steps corresponds to the *sequential* analysis of each circuit, which are independent of each other and could be parallelized. Despite this, the performance of the tool was considered to be adequate, given the strong support in pinpointing modeling errors.

Verification. We verified the obtained SMV models against a set of domain dependent specifications. For RRIS `r-switch` we consider 16 safety properties describing how the switch changes in response to commands, for both the nominal and faulty stub modalities. RRIS `routes48` implements a controlling logic which avoids that two incompatible routes are enabled simultaneously: incompatibilities are checked with a system of lockings of the railway switches shared by concurrent routes. An *incompatibility table* represents which pairs of routes can be sequentially activated. The table is not symmetric, as the incompatibility relation depends on the activation order. Thanks to the simplifications, we were able to model check all the 2256 entries, proving both safety (incompatible pairs) and liveness (compatible pairs) properties.

Fig. 5 shows the impact of the simplification steps on the model checking times for the verification of `r-switch` (on the left hand side) and `routes48` (on the right hand side). For the latter, we consider 105 queries tackled with IC3

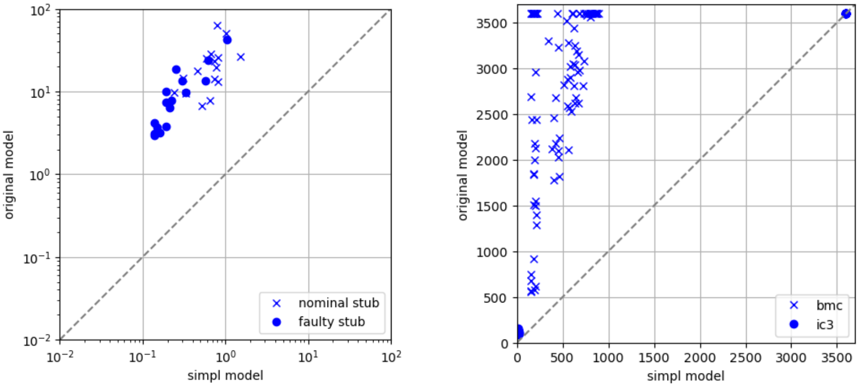


Fig. 5. Effect of the simplifications on the model checking times.

and 91 queries tackled with BMC under 1 hour timeout. Only 17 IC3 instances encountered this threshold with the simplified model, against the 50 time outs of the original model, for both IC3 and BMC.

9 Conclusions

In this paper we presented Norma, a tool for the modeling and formal analysis of relay-based railway interlocking systems (RRIS). Norma allows to graphically represent the wide class of RRIS of the Italian railway network, and provides various checks to ease the task of the modeler. Furthermore, it provides an optimized compilation to the input language of timed nuXmv, that converts the RRIS into a symbolic infinite-state timed transition system. This enables simulation and effective model checking of temporal properties. The experimental results clearly demonstrate the effectiveness of the simplification techniques. The tool is also shown to provide strong feedback to the user to support debugging in the modeling process.

Norma is being extensively used within RFI. Despite the support provided to the modelers, the sheer size of real-world RRIS results in a very high human modeling effort. We are currently experimenting with deep learning techniques to automate – at least partially – the modeling step. In this setting, the formal analysis capabilities will be fundamental to detect misclassified samples. In the future, we will work on obtaining high-coverage test suites for RRIS, to improve testing of computer-based RIS. We will also explore compositional contract-based reasoning to reduce the computation time of model checking, and provide clear interfaces between RRIS modules. A tighter integration of the simulation capabilities within the tool front-end is also planned.

References

1. Git: A free and open source distributed version control system. https://git-scm.com/
2. Dia: A GTK+ based diagram creation program. https://gitlab.gnome.org/GNOME/dia
3. Poppler: a PDF rendering library. https://poppler.freedesktop.org/
4. GitLab: A web-based DevOps lifecycle tool. https://gitlab.com/
5. De Almeida Pereira, D.I., Déharbe, D., Perin, M., Bon, P.: B-specification of relay-based railway interlocking systems based on the propositional logic of the system state evolution. In: RSSRail. Lecture Notes in Computer Science, vol. 11495, pp. 242–258. Springer (2019)
6. Amendola, A., Becchi, A., Cavada, R., Cimatti, A., Griggio, A., Scaglione, G., Susi, A., Tacchella, A., Tessi, M.: A model-based approach to the design, verification and deployment of railway interlocking system. In: ISoLA (3). Lecture Notes in Computer Science, vol. 12478, pp. 240–254. Springer (2020)
7. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 334–342. Springer International Publishing, Cham (2014)
8. Cavada, R., Cimatti, A., Mover, S., Sessa, M., Cadavero, G., Scaglione, G.: Analysis of relay interlocking systems via SMT-based model checking of switched multi-domain kirchhoff networks. In: FMCAD. pp. 1–9. IEEE (2018)
9. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with timed transition systems and timed temporal properties. In: CAV (1). Lecture Notes in Computer Science, vol. 11561, pp. 376–386. Springer (2019)
10. Gario, M., Micheli, A., Kessler, F.B.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms
11. Haxthausen, A.E., Kjær, A.A., Bliguet, M.L.: Formal development of a tool for automated modelling and verification of relay interlocking systems. In: FM. Lecture Notes in Computer Science, vol. 6664, pp. 118–132. Springer (2011)
12. James, P., Moller, F., Nga, N.H., Roggenbach, M., Schneider, S.A., Treharne, H.: Techniques for modelling and verifying railway interlockings. Int. J. Softw. Tools Technol. Transf. **16**(6), 685–711 (2014)
13. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: CAV. Lecture Notes in Computer Science, vol. 4144, pp. 424–437. Springer (2006)
14. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. **36**(5), 450–462 (1993)
15. Sun, P., Dutilleul, S.C., Bon, P.: A model pattern of railway interlocking system by Petri nets. In: MT-ITS. pp. 442–449. IEEE (2015)

Efficient Neural Network Analysis with Sum-of-Infeasibilities

Haoze Wu[1]✉, Aleksandar Zeljić[1], Guy Katz[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
[2] The Hebrew University of Jerusalem, Jerusalem, Israel

Abstract. Inspired by sum-of-infeasibilities methods in convex optimization, we propose a novel procedure for analyzing verification queries on neural networks with piecewise-linear activation functions. Given a convex relaxation which over-approximates the non-convex activation functions, we encode the violations of activation functions as a cost function and optimize it with respect to the convex relaxation. The cost function, referred to as the Sum-of-Infeasibilities (SoI), is designed so that its minimum is zero and achieved only if all the activation functions are satisfied. We propose a stochastic procedure, `DeepSoI`, to efficiently minimize the SoI. An extension to a canonical case-analysis-based complete search procedure can be achieved by replacing the convex procedure executed at each search state with `DeepSoI`. Extending the complete search with `DeepSoI` achieves multiple simultaneous goals: 1) it guides the search towards a counter-example; 2) it enables more informed branching decisions; and 3) it creates additional opportunities for bound derivation. An extensive evaluation across different benchmarks and solvers demonstrates the benefit of the proposed techniques. In particular, we demonstrate that SoI significantly improves the performance of an existing complete search procedure. Moreover, the SoI-based implementation outperforms other state-of-the-art complete verifiers. We also show that our technique can efficiently improve upon the perturbation bound derived by a recent adversarial attack algorithm.

Keywords: neural networks · sum of infeasibilities · convex optimization · stochastic local search.

1 Introduction

Neural networks have become state-of-the-art solutions in various application domains, e.g., face recognition, voice recognition, game-playing, and automated piloting [47,30,55,7]. While generally successful, neural networks are known to be susceptible to input perturbations that humans are naturally invariant to [61,41]. This calls the trustworthiness of neural networks into question, particularly in safety-critical domains.

In recent years, there has been a growing interest in applying formal methods to neural networks to analyze certain robustness or safety *specifications* [43]. Such specifications are often defined by a collection of partial input/output relations:

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 143–163, 2022.
https://doi.org/10.1007/978-3-030-99524-9_8

e.g., the network uniformly and correctly classifies inputs within a certain distance (in some l_p norm) of a selection of input points. The goal of formal verification is to either prove that the network meets the specification or to disprove it by constructing a counter-example.

Most standard activation functions in neural networks are non-linear, making them challenging to reason about. Consider the rectified linear unit (ReLU): if a ReLU can take both positive and negative inputs, a verifier will typically need to consider, separately, each of these two activation phases. Naive case analysis requires exploring a number of combinations that is exponential in the number of ReLUs, which quickly becomes computationally infeasible for large networks. To mitigate this complexity, neural network verifiers typically operate on convex relaxations of the activation functions. The relaxed problem can often be solved with an efficient convex procedure, such as Simplex [35,23] or (sub-)gradient methods [51,21]. Due to the relaxation, however, a solution may be inconsistent with the true activation functions. When this happens, the convex procedure cannot make further progress on its own. For this reason, to ensure completeness, the convex procedure is typically embedded in an exhaustive search shell, which encodes the activation functions explicitly and branches on them when needed. While the exhaustive search ensures progress, it also brings back the problem of combinatorial explosion. This raises the key question: **can we guide the convex procedure to satisfy the activation functions without explicitly encoding them?**

In convex optimization, the sum-of-infeasibilities (SoI) [10] function measures the error (with respect to variable bounds) of a variable assignment. Minimizing the SoI naturally guides the procedure to a satisfying assignment. In this paper, we extend this idea to instead represent the error in the non-linear activation functions. The goal is to "softly" guide the search over the relaxed problem using information about the precise activation functions. If an assignment is found for which the SoI is zero, then not only is the assignment a solution for the relaxation, but it also solves the precise problem. Encoding the SoI w.r.t. the piecewise-linear activation functions yields a concave piecewise-linear function, which is challenging to minimize directly. Instead, we propose to minimize the SoI for individual *activation patterns* and reduce the SoI minimization to a *stochastic* search for the activation pattern where the SoI is minimal. The advantage is that for each activation pattern, the SoI collapses into a *linear* cost function, which can be easily handled by a convex solver. We introduce a specialized procedure, `DeepSoI`, which uses Markov chain Monte Carlo (MCMC) search to efficiently navigate towards activation patterns at the global minimum of the SoI. If the minimal SoI is ever zero for an activation pattern, then a solution has been found.

An extension to a canonical complete search procedure can be achieved by replacing the convex procedure call at each search state with the `DeepSoI` procedure. Since the SoI contains additional information about the problem, we propose a novel SoI-aware branching heuristic based on the estimated impact of each activation function on the SoI. Finally, `DeepSoI` naturally preserves new bounds derived during the execution of the underlying convex procedure (e.g., Simplex), which further prunes the search space in the complete search. For simplicity, we focus on ReLU activation functions in this paper, though the proposed approach can be applied to any piecewise-linear activation function.

We implemented the proposed techniques in the Marabou framework for Neural Network Analysis [36] and performed an extensive performance evaluation on a wide range of benchmarks. We compare against multiple baselines and show that extending a complete search procedure with our SoI-based techniques results in significant overall speed-ups. Finally, we present an interesting use case for our procedure — efficiently improving the perturbation bounds found by AutoAttack [17], a state-of-the-art adversarial attack algorithm.

To summarize, the contributions of the paper are: (i) a technique for guiding a convex solver with an SoI function w.r.t. the activation functions; (ii) `DeepSoI`—a procedure for minimizing the non-linear SoI via the interleaving use of an MCMC sampler and a convex solver; (iii) an SoI-aware branching heuristic, which complements the integration of `DeepSoI` into a case-analysis based search shell; and (iv) a thorough evaluation of the proposed techniques.

The rest of the paper is organized as follows. Section 2 presents an overview of related work. Section 3 introduces preliminaries. Section 4 introduces the SoI and proposes a solution for its minimization. Section 5 presents the analysis procedure `DeepSoI`, its use in the complete verification setting, and an SoI-aware branching heuristic. Section 6 presents an extensive experimental evaluation. Conclusions and future work are in Section 7.

2 Related Work

Approaches to complete analysis of neural networks can be divided into SMT-based [35,36,23], reachability-analysis based [5,64,65,29,25], and the more general branch-and-bound approaches [1,63,24,44,13,37,9]. As mentioned in [14], these approaches are related, and differ primarily in their techniques for bounding and branching. Given the computational complexity of neural network verification, a diverse set of research directions aims to improve performance in practice. Many approaches prune the search space using tighter convex relaxations and bound inference techniques [64,23,31,58,56,45,76,70,67,66,20,63,69,52,62,51,73,26,68,59,8,57]. Another direction leverages parallelism by exploiting independent structures in the search space [48,75,71]. Different encodings of the neural network verification problems have also been studied: e.g., as MILP problems that can be tackled by off-the-shelf solvers [63,2], or as dual problems admitting efficient GPU-based algorithms [12,21,22,19]. `DeepSoI` can be instantiated with any sound convex relaxations and matching convex procedures. It can also be installed in any case-analysis-based complete search shell, therefore integrating easily with existing parallelization techniques, bound-tightening passes, and branching heuristics.

Two approaches most relevant to our work are Reluplex [35] and PeregriNN [37]. Reluplex invokes an LP solver to solve the relaxed problem, and then updates its solution to satisfy the violated activation functions — with the hope of nudging the produced solutions towards a satisfying assignment. However, the updated solution by Reluplex could violate the linear relaxation, leading to non-convergent cycling between solution updates and LP solver calls, which can only be broken by branching. In contrast, our approach uses information about the precise activation functions to *actively* guide the convex solver. Furthermore, in the limit `DeepSoI` converges to a solution (if one exists). PeregriNN also uses an objective function to guide the solving of the convex relaxation. However, their objective function *approximates* the ReLU violation and does not guarantee a real counter-example when the minimum is reached. In contrast, the SoI function captures the *exact* ReLU violation, and if a zero-valued point is found, it is *guaranteed* to be a real counter-example. We compare our techniques to PeregriNN in Section 6.

We use MCMC-sampling combined with a convex procedure to minimize the concave piecewise-linear SoI function. MCMC-sampling is a common approach for stochastically minimizing irregular cost functions that are not amenable to exact optimization techniques [32,53,3]. Other stochastic local search techniques [54,27] could also be used for this task. However, we chose MCMC because it is adept at escaping local optima, and in the limit, it samples more frequently the region around the optimum value. As one point of comparison, in Section 6, we compare MCMC-sampling with a Walksat-based [54] local search strategy.

3 Preliminaries

Neural Networks. We define a feed-forward, convolutional, or residual neural network with $k+1$ layers as a set of *neurons* N, topologically ordered into *layers* $L_0, ..., L_k$, where L_0 is the input layer and L_k is the output layer. Given $n_i, n_j \in N$, we use $n_i \prec n_j$ to denote that the layer of n_i precedes the layer of n_j. The value of a neuron $n_i \in N \backslash L_0$ is computed as $act_i(b_i + \sum_{n_j \prec n_i} w_{ij} * n_j)$, an affine transformation of the preceding neurons followed by an activation function act_i. We use n_i^b and n_i^a to represent the pre- and post-activation values of such a neuron: $n_i^a = act_i(n_i^b)$. For $n_i \in L_0$, n_i^b is undefined and we assume n_i^a can take any value. In this paper, we focus on ReLU neural networks. That is, act_i is the ReLU function ($ReLU(x) = \max(0, x)$) unless n_i belongs to the output layer L_k, in which case act_i is the identity function. We use $R(N)$ to denote the set of ReLU neurons in N. An *activation pattern* is defined by choosing a particular phase (either active or inactive) for every $n \in R(N)$ (i.e., choosing either $n_i^b < 0$ or $n_i^b \geq 0$ for each $n_i \in R(N)$).

Neural Network Verification as Satisfiability. Consider the verification of a property P over a neural network N. The property P has the form $P_{in} \Rightarrow P_{out}$, where P_{in} and P_{out} constrain the input and output layers, respectively. P states that for each input point satisfying P_{in}, the output layer satisfies P_{out}. To formalize the verification problem, we first define the set of *variables* in a neural network N, denoted as $Var(N)$, to be $\cup_{n_i \in N \backslash L_k} \{n_i^a\} \cup \cup_{n_i \in N \backslash L_0} \{n_i^b\}$. We define a *variable assignment*, $\alpha : Var(N) \rightarrow \mathbb{R}$, to be a mapping from variables in N to real values. The verification task thus can be formally stated as finding a variable assignment α that satisfies the following set of *constraints* over $Var(N)$ (denoted as ϕ):[3]

$$\forall n_i \in N \backslash L_0, n_i^b = b_i + \sum_{n_j \prec n_i} w_{ij} * n_j^a \tag{1a}$$

$$\forall n_i \in R(N), n_i^a = act_i(n_i^b) \tag{1b}$$

$$P_{in} \wedge \neg P_{out} \tag{1c}$$

If such an assignment α exists, we say that ϕ is *satisfiable* and can conclude that P does not hold, as from α we can retrieve an input $x \in P_{in}$, such that the neural network's output violates P_{out}. If such an α does *not* exist, we say ϕ is *unsatisfiable* and can conclude that P holds. We use $\alpha \models \phi$ to denote that an assignment α satisfies ϕ. In short, verifying whether P holds on a neural network N boils down to deciding the satisfiability of ϕ. We refer to ϕ also as the *verification query* in this paper.

Convex Relaxation of Neural Networks. Deciding whether P holds on a ReLU network N is NP-complete [35]. To curb intractability, many verifiers consider the convex (e.g., linear, semi-definite) relaxation of the verification problem, sacrificing completeness in exchange for a reduction in the computational complexity. We use $\widetilde{\phi}$ to denote the convex relaxation of the exact problem ϕ. If $\widetilde{\phi}$ is unsatisfiable, then ϕ is also unsatisfiable, and property P holds. If the convex relaxation is satisfiable with satisfying assignment α and α also satisfies ϕ, then P does not hold.

In this paper, we use the *Planet relaxation* introduced in [23]. It is a linear relaxation, illustrated in Figure 1. Each ReLU constraint $\text{ReLU}(n^b) = n^a$ is over-approximated by three linear constraints: $n^a \geq 0$, $n^a \geq n^b$, and $n^a \leq \frac{u}{u-l}n^b - \frac{u*l}{u-l}$, where u and l are the upper and lower bounds of n^b, respectively (which can be derived using bound-tightening techniques such as those in [67,58,76]). If Constraint 1c is also linear, the convex relaxation $\widetilde{\phi}$ is a Linear Program, whose satisfiability can be decided efficiently (e.g., using the *Simplex* algorithm [18]).

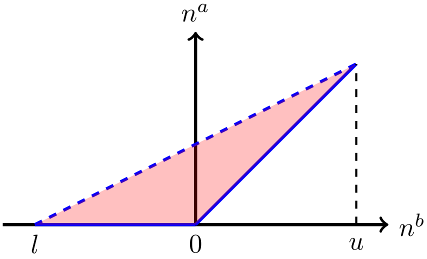


Fig. 1: The Planet relaxation.

Sum-of-Infeasibilities. In convex optimization [10,39], the sum-of-infeasibilities (SOI) method can be used to direct the feasibility search. The satisfiability of a formula ϕ is cast as an optimization problem, with an objective function representing the total *error* (i.e., the sum of the distances from each out-of-bounds variable to its closest bound). The lower bound of f is 0 and is achieved only if ϕ is satisfiable. In our context, we use a similar function f_{soi}, but with the difference that it represents the total error of the ReLU constraints in ϕ. In our case, f_{soi} is non-convex, and thus a more sophisticated approach is needed to minimize it efficiently.

[3] The verification can also be equivalently viewed as an optimization problem [14].

Complete Analysis via Exhaustive Search. One common approach for complete verification involves constructing a search tree and calling a *convex procedure* solveConv at each tree node, as shown in Algorithm 1. solveConv solves the convex relaxation $\widetilde{\phi}$ and returns a pair r, α where either: 1) $r = \texttt{SAT}$ and $\alpha \models \widetilde{\phi}$; or 2) $r = \texttt{UNSAT}$ and $\widetilde{\phi}$ is unsatisfiable. If $\widetilde{\phi}$ is unsatisfiable or α also satisfies ϕ, then the result for $\widetilde{\phi}$ also holds for ϕ and is returned. Otherwise, the search space is divided further using branch, which returns a set Ψ of sub-problems such that ϕ and $\bigvee \Psi$ are equisatisfiable.

Algorithm 1 Complete search.

```
Input: a verification query $\phi$.
Output: SAT/UNSAT
function completeSearch($\phi$)
    $\phi \leftarrow$ tightenBounds($\phi$)
    $r, \alpha \leftarrow$ solveConv($\widetilde{\phi}$)
    if $r = \texttt{UNSAT} \vee \alpha \models \phi$ then
        return $r$
    for $\phi_i \in$ branch($\phi$) do
        if completeSearch($\phi_i$) = SAT then
            return SAT
    return UNSAT
```

Before invoking solveConv to solve $\widetilde{\phi}$, it is common to first call an efficient bound-tightening procedure (tightenBounds) to prune the search space or even derive UNSAT preemptively. This tightenBounds procedure can be instantiated in various ways, including with analyses based on LiPRA [74,76,58,70], kReLU [56], or PRIMA [49]. In addition to the dedicated bound-tightening pass, some convex procedures (e.g., Simplex) also naturally lend themselves to bound inference during their executions [38,35]. The overall performance of Algorithm 1 depends on the efficacy of bound-tightening, the branching heuristics, and the underlying convex procedure.

Adversarial attacks. Adversarial attacks [61,46,28,15] are another approach for assessing neural network robustness. While verification uses exhaustive search to either prove or disprove a particular specification, adversarial attacks focus on efficient heuristic algorithms for the latter. From another perspective, they can demonstrate *upper bounds* on neural network robustness. In Section 6, we show that our analysis procedure can improve the bounds found by AutoAttack [17].

4 Sum of Infeasibilities in Neural Network Analysis

In this section, we introduce our SoI function, consider the challenge of its minimization, and present a stochastic local search solution.

4.1 The Sum of Infeasibilities

As mentioned above, in convex optimization, an SoI function represents the sum of errors in a candidate variable assignment. Here, we build on this idea by introducing a cost function f_{soi}, which computes the sum of errors introduced by a convex relaxation of a verification query. We aim to use f_{soi} to reduce the satisfiability problem for ϕ to a simpler optimization problem. We will need the following property to hold.

Condition 1. *For an assignment α, $\alpha \models \phi$ iff $\alpha \models \widetilde{\phi} \wedge f_{soi} \leq 0$.*

If Condition 1 is met, then satisfiability of ϕ reduces to the following minimization problem:

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & f_{soi} \\ \text{subject to} \quad & \alpha \models \widetilde{\phi} \end{aligned} \tag{2}$$

To formulate the SoI for ReLU networks, we first define the error in a ReLU constraint n as:

$$\mathbf{E}(n) = \min(n^a - n^b, n^a) \tag{3}$$

The two arguments correspond to the error when the ReLU is in the active and inactive phase, respectively. Recall that the Planet relaxation constrains (n^b, n^a) in the triangular area in Figure 1, where $n^a \geq n^b$ and $n^a \geq 0$. Thus, the minimum of $\mathbf{E}(n)$ subject to $\widetilde{\phi}$ is non-negative, and furthermore, $\mathbf{E}(n) = 0$ iff the ReLU constraint n is satisfied (this is also true for any relaxation at least as tight as the Planet relaxation). We now define f_{soi} as the sum of errors in individual ReLUs:

$$f_{soi} = \sum_{n \in R(N)} \mathbf{E}(n) \tag{4}$$

Theorem 1. *Let N be a set of neurons for a neural network, ϕ a verification query (an instance of (1)), and $\widetilde{\phi}$ the planet relaxation of ϕ. Then f_{soi} as given by (4) satisfies Condition 1.*

Proof. It is straightforward to show that f_{soi} subject to $\widetilde{\phi}$ is non-negative and is zero if and only if each $\mathbf{E}(n_i)$ is zero. That is, $\min f_{soi}$ subject to $\widetilde{\phi}$ is zero if and only if all ReLUs are satisfied. Therefore, if α satisfies ϕ, then $\alpha \models f_{soi} = 0$. On the other hand, since an assignment α that satisfies $\widetilde{\phi}$ can only violate the ReLU constraints in ϕ, if $\alpha \models f_{soi} = 0$, then all the constraints in ϕ must be satisfied, i.e., $\alpha \models \phi$. □

Note that the error $\mathbf{E}$, and its extension to SoI, can easily be defined for other piecewise-linear functions besides ReLU. We now turn to the question of minimizing f_{soi}. Observe that

$$\min f_{soi} = \min \sum_{n \in R(N)} \mathbf{E}(n) = \min \Big(\{ f \mid f = \sum_{n_i \in R(N)} t_i, \quad t_i \in \{n_i^a - n_i^b, n_i^a\} \} \Big). \tag{5}$$

Thus, f_{soi} is the minimum over a set, which we will denote S_{soi}, of linear functions. Although $\min f_{soi}$ cannot be used directly as an objective in a convex procedure, we could minimize each individual linear function $f \in S_{soi}$ with a convex procedure and then keep the minimum over all functions. We refer to the functions in S_{soi} as *phase patterns* of f_{soi}. For notational convenience, we define *cost*(f, ϕ) to be the minimum of f subject to ϕ. The minimization problem (2) can thus be restated as searching for the phase pattern $f \in S_{soi}$, where *cost*$(f, \widetilde{\phi})$ is minimal. Note that for a particular *activation pattern*, $f_{soi} = f$ for some $f \in S_{soi}$. From this perspective, searching for the $f \in S_{soi}$ where *cost*$(f, \widetilde{\phi})$ is minimal can also be viewed as searching for the activation pattern where the global minimum of f_{soi} is achieved.

4.2 Stochastically Minimizing the SoI with MCMC Sampling

In the worst case, finding the minimal value of $cost(f, \widetilde{\phi})$ requires enumerating and minimizing each f in S_{soi} (or equivalently, minimizing f_{soi} for each activation pattern), which has size $2^{|R(N)|}$. However, importantly, the search can terminate immediately if a phase pattern f is found such that $cost(f, \widetilde{\phi}) = 0$. We leverage this fact below. Note that each phase pattern has $|R(N)|$ adjacent phase patterns, each differing in only one linear subexpression. The space of phase patterns is thus fairly dense, making it amenable to traversal using stochastic local search methods. In particular, intelligent hill-climbing algorithms, which can be made robust against local optima, are well suited for this task.

Markov chain Monte Carlo (MCMC) [11] methods are such an approach. In our context, MCMC methods can be used to generate a sequence of phase patterns $f_0, f_1, f_2 ... \in S_{soi}$, with the desirable property that in the limit, the phase patterns are more frequently from the minimum region of $cost(f, \widetilde{\phi})$.

We use the Metropolis-Hastings (M-H) algorithm [16], a widely applicable MCMC method, to construct the sequence. The algorithm maintains a current phase pattern f and proposes to replace f with a new phase pattern f'. The proposal comes from a *proposal distribution* $q(f'|f)$ and is accepted with a certain *acceptance probability* $m(f \to f')$. If the proposal is accepted, f' becomes the new current phase pattern. Otherwise, another proposal is considered. This process is repeated until one of the following scenarios happen: 1) a phase pattern f is chosen with $cost(f, \widetilde{\phi}) = 0$; 2) a predetermined computational budget is exhausted; or 3) all possible phase patterns have been considered. The last scenario is generally infeasible for non-trivial networks. In order to employ the algorithm, we transform $cost(f, \widetilde{\phi})$ into a probability distribution $p(f)$ using a common method [34]:

$$p(f) \propto \exp(-\beta \cdot cost(f, \widetilde{\phi}))$$

where β is a configurable parameter. If the proposal distribution is symmetric (i.e., $q(f|f') = q(f'|f)$), the acceptance probability is the following (often referred to as the *Metropolis ratio*) [34]:

$$m(f \to f') = \min(1, \frac{p(f')}{p(f)}) = \min\Big(1, \exp\big(-\beta \cdot \big(cost(f', \widetilde{\phi}) - cost(f, \widetilde{\phi})\big)\big)\Big)$$

Importantly, under this acceptance probability, *a proposal reducing the value of the cost function is always accepted, while a proposal that does not may still be accepted* (albeit with a probability that is inversely correlated with the increase in the cost). This means that the algorithm always greedily moves to a lower cost phase pattern whenever it can, but it also has an effective means for escaping local minima. Note that since the sample space is finite, as long as the proposal strategy is *ergodic*,[4] in the limit, the probability of sampling *every* phase pattern (therefore deciding the satisfiability of ϕ) converges to 1. However, we do not

[4] A proposal strategy is ergodic if it is capable of transforming any phase pattern to any other through a sequence of applications. We use a symmetric and ergodic proposal distribution as explained in Section 5.1.

have formal guarantees about the convergence rate, and it is usually impractical to prove unsatisfiability this way. Instead, as we shall see in the next section, we enable complete verification by embedding the M-H algorithm in an exhaustive search shell.

5 The DeepSoI Algorithm

In this section, we introduce `DeepSoI`, a novel verification algorithm that leverages the SoI function, and show how to integrate it with a complete verification procedure. We also discuss the impact of `DeepSoI` on complete verification and propose an SoI-aware branching heuristic.

5.1 DeepSoI

Algorithm 2 Analyzing ϕ with `DeepSoI`.

```
 1: Input: A verification query $\phi$.
 2: Output: SAT/UNSAT/UNKNOWN
 3: function DeepSoI($\phi$)
 4:     $r, \alpha_0 \leftarrow$ solveConv($\widetilde{\phi}$)
 5:     if $r =$ UNSAT $\vee \alpha_0 \models \phi$ then return $r, \alpha_0$          } Phs. I
 6:     $k, f \leftarrow 0,$ initPhasePattern($\alpha_0$)
 7:     $\alpha, c \leftarrow$ optimizeConv($f, \widetilde{\phi}$)
 8:     while $c > 0 \wedge \neg$exhausted() $\wedge k < T$ do
 9:         $f' \leftarrow$ propose($f$)
10:         $\alpha', c' \leftarrow$ optimizeConv($f', \widetilde{\phi}$)                   } Phs. II
11:         if accept($c, c'$) then $f, c, \alpha \leftarrow f', c', \alpha'$
12:         else $k \leftarrow k + 1$
13:     if $c = 0$ then return SAT, $\alpha$
14:     else return exhausted() ? UNSAT : UNKNOWN
```

Our procedure `DeepSoI`, shown in Algorithm 2, takes an input verification query ϕ and tries to determine its satisfiability. `DeepSoI` follows the standard two-phase convex optimization approach. Phase I finds *some* assignment α_0 satisfying $\widetilde{\phi}$, and phase II attempts to optimize the assignment using the M-H algorithm. Phase II uses a convex optimization procedure optimizeConv which takes an objective function f and a formula ϕ as inputs and returns a pair α, c, where $\alpha \models \phi$ and $c = \mathit{cost}(f, \phi)$ is the optimal value of f. Phase II chooses an initial phase pattern f based on α_0 (Line 6) and computes its optimal value c. The M-H algorithm repeatedly proposes a new phase pattern f' (Line 9), computes its optimal value c', and decides whether to accept f' as the current phase pattern f. The procedure returns `SAT` when a phase pattern f is found such that $\mathit{cost}(f, \widetilde{\phi}) = 0$ and `UNSAT` if all phase patterns have been considered (exhausted returns true) before a threshold of T rejections is exceeded. Otherwise, the analysis is inconclusive (`UNKNOWN`).

The accept method decides whether a proposal is accepted based on the Metropolis ratio (see Section 4). Function initPhasePattern proposes the initial phase pattern f induced by the activation pattern corresponding to assignment α_0. Our proposal strategy (propose) is also simple: pick a ReLU n at random and flip its cost component in the current phase pattern f (either from $n^a - n^b$ to n^a, or vice-versa). This proposal strategy is symmetric, ergodic, and performs well in practice. Both the initialization strategy and the proposal strategy are crucial to the performance of the M-H Algorithm, and exploring more sophisticated strategies is a promising avenue for future work. Importantly, the same convex procedure is used in both phases. Therefore, from the perspective of the convex procedure, `DeepSoI` solves a sequence of convex optimization problems that differ only in the objective functions, and each problem can be solved incrementally by updating the phase pattern without the need for a restart.

5.2 Complete Analysis and Pseudo-impact Branching

To extend a canonical complete verification procedure (i.e., Algorithm 1), its solveConv call is replaced with `DeepSoI`. Note that the implementation of branch in this algorithm has a significant influence on its performance. Here, we consider an SoI-aware implementation of branch, which makes decisions by selecting a particular ReLU to be active or inactive. The choice of *which* ReLU is crucial. Intuitively, we want to branch on the ReLU with the most impact on the value of f_{soi}. After branching, `DeepSoI` should be closer to either: finding a satisfying assignment (if f_{soi} is decreased), or determining unsatisfiability (if f_{soi} is increased). Computing the exact impact of each ReLU n on f_{soi} would be expensive; however, we can estimate it by recording changes in f_{soi} during the execution of `DeepSoI`.

Concretely, for each ReLU n, we maintain its *pseudo-impact*,[5] $\mathbf{PI}(n)$, which represents the estimated impact of n on f_{soi}. For each n, $\mathbf{PI}(n)$ is initialized to 0. Then during the M-H algorithm, whenever the next proposal flips the cost component of ReLU n, we calculate the local impact on f_{soi}: $\Delta = |cost(f, \widetilde{\phi}) - cost(f', \widetilde{\phi})|$. We use Δ to update the value of $\mathbf{PI}(n)$ according to the *exponential moving average* (EMA): $\mathbf{PI}(n) = \gamma * \mathbf{PI}(n) + (1-\gamma) \cdot \Delta$, where γ attenuates previous estimates of n's impact. We use EMA because recent estimates are more likely to be relevant to the current phase pattern. At branching time, the *pseudo-impact heuristic* picks $\arg\max_n \mathbf{PI}(n)$ as the ReLU to split on. The heuristic is inaccurate early in the search, so we use a static branching order (e.g., [71,13]) while the depth of the search tree is shallow (e.g., < 3).

6 Experimental Evaluation

In this section, we present an experimental evaluation of the proposed techniques. Our experiments include: 1. an ablation study to examine the effect of each proposed technique; 2. a run-time comparison of our prototype with other complete analyzers; 3. an empirical study of the choice of the rejection threshold T in Algorithm 2; and 4. an experiment in using our analysis procedure to improve the perturbation bounds found by AutoAttack [17], an adversarial attack algorithm. An artifact with which the results can be replicated is available on Zenodo [72].

[5] The name is in analogy to pseudo-cost branching heuristics in MILP, where the integer variable with the largest impact on the objective function is chosen [6].

6.1 Implementation.

We implemented our techniques in Marabou [36], an open-source toolbox for analyzing Neural Networks. It features a user-friendly python API for defining properties and loading networks, and a native implementation of the Simplex algorithm. Besides the Markov chain Monte Carlo stochastic local search algorithm presented in Section 5.1 and the pseudo-impact branching heuristic presented in Section 5.2, we also implemented a Walksat-inspired [54] stochastic local search strategy to evaluate the effectiveness of MCMC-sampling as a local minimization strategy. Concretely, from a phase pattern f, the strategy greedily moves to a neighbor f' of f, with $\mathit{cost}(f', \widetilde{\phi}) < \mathit{cost}(f, \widetilde{\phi})$. If no such f' exists (i.e., a local minimum has been reached), the strategy moves to a random neighbor.

The solveConv and optimizeConv methods in Algorithm 2 can be instantiated with either the native Simplex engine of Marabou or with Gurobi, an off-the-shelf (MI)LP-solver. The tightenBounds method is instantiated with the DeepPoly analysis from [58], an effective and light-weight bound-tightening pass, which is also implemented in Marabou.

6.2 Benchmarks.

We evaluate on networks from four different applications: MNIST, CIFAR10, TaxiNet, and GTSR. The network architectures are shown in Table 2.

Bench.	Model	Type	ReLUs	Hid. Layers
MNIST	MNIST$_1$	FC	512	2
	MNIST$_2$	FC	1024	4
	MNIST$_3$	FC	1536	6
TaxiNet	Taxi1	Conv	688	6
	Taxi2	Conv	2048	4
	Taxi3	Conv	2752	6
CIFAR10	CIFAR10$_b$	Conv	1226	4
	CIFAR10$_w$	Conv	4804	4
	CIFAR10$_d$	Conv	5196	6
GTSR	GTSR$_1$	FC	600	3
	GTSR$_2$	Conv	2784	4

Fig. 2: Architecture overview.

The MNIST [42] and CIFAR10 [40] networks are established benchmarks used in previous literature (e.g., [19,37,71,75]) as well as in the 2021 VNN Competition [4]. Notably, the same MNIST networks are used to evaluate the original PeregriNN work.

For MNIST and CIFAR10 networks, we check robustness against targeted l_∞ attacks on randomly selected images from the test sets. The target labels are chosen randomly from the incorrect labels, and the perturbation bound is sampled uniformly from $\{0.01, 0.03, 0.06, 0.09, 0.12, 0.15\}$. The TaxiNet [33] benchmark set comprises robustness queries over regression models used for vision-based autonomous taxiing. Given an image of the taxiway captured by the aircraft, the model predicts its displacement (in meters) from the center of the taxiway. A controller uses the output to adjust the heading of the aircraft. Robustness is parametrized by input perturbation δ and output perturbation ϵ; we sample (δ, ϵ) uniformly from $\{0.01, 0.03, 0.06\} \times \{2, 6\}$. The GTSR benchmark set comprises robustness queries on image classifiers trained on a subset of the German Traffic Sign Recognition benchmark set [60]. Given a 32×32 RGB image the networks classify it as one of seven different kinds of traffic signs. A hazing perturbation [50] drains color from the image to create a veil of colored mist. Given an image I, a

Bench. (#)	$\texttt{MILP}_{\texttt{MIPVerify}}$		$\texttt{LP}^{\texttt{snc}}$		$\texttt{SOI}^{\texttt{snc}}_{\texttt{mcmc}}$		$\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$		$\texttt{SOI}^{\texttt{pi}}_{\texttt{wsat}}$	
	Solv.	Time	Solv.	Time	Solv.	Time	Solv.	Time	Solv.	Time
MNIST$_1$ (90)	**77**	19791	47	6892	66	5635	70	5976	68	5388
MNIST$_2$ (90)	29	6125	24	514	**36**	4356	31	757	31	909
MNIST$_3$ (90)	23	957	21	1609	34	9519	**35**	8327	33	5270
Taxi$_1$ (90)	**90**	786	61	9054	80	4257	89	1390	90	1489
Taxi$_2$ (90)	40	17093	2	891	70	5503	**71**	6889	71	7407
Taxi$_3$ (90)	**89**	5058	64	69715	87	1034	88	2164	87	997
CIFAR10$_b$ (90)	**76**	4316	26	7425	69	6286	73	16469	69	5200
CIFAR10$_w$ (90)	38	9879	18	845	41	4619	42	8129	**42**	6415
CIFAR10$_d$ (90)	30	4198	21	3395	51	17679	51	15056	51	15015
GTSR$_1$ (90)	90	2541	**90**	2435	89	4900	90	15238	90	4805
GTSR$_2$ (90)	90	23613	**90**	4456	90	7507	90	10426	90	6180
Total (990)	673	94354	463	107230	711	71294	**730**	90822	721	59073

Table 1: Instances solved by different configurations and their runtime (in seconds) on solved instances.

perturbation parameter ϵ, and a haze color C^f, the perturbed image I' is equal to $(1-\epsilon)\cdot I + \epsilon \cdot C^f$. The robustness queries check whether the bound yielded by the test-based method in [50] is minimal.All pixel values are normalized to $[0, 1]$, and the chosen perturbation values yield a mix of non-trivial SAT and UNSAT instances.

6.3 Experimental Setup.

Experiments are run on a cluster equipped with Intel Xeon E5-2637 v4 CPUs running Ubuntu 16.04. Unless specified otherwise, each job is run with 1 thread, 8GB memory, and a 1-hour CPU timeout. By default, the solveConv and optimizeConv methods use Gurobi. The following hyper-parameters are used: the rejection threshold T in Algorithm 2 is 2; the discount factor γ in the EMA is 0.5; and the probability density parameter β in the Metropolis ratio is 10. These parameters are empirically optimal on a subset of MNIST benchmarks. In practice, the performance is most sensitive to the rejection threshold T, and below (Section 6.6), we conduct experiments to study its effect.

6.4 Ablation study of the proposed techniques.

To evaluate each individual component of our proposed techniques, we run several configurations across the full set of benchmarks described above.

We first consider two baselines that do not minimize the SoI: 1. $\texttt{LP}^{\texttt{snc}}$— runs Algorithm 1 with the Split-and-Conquer (SnC) branching heuristic [71], which estimates the number of tightened bounds from a ReLU split; 2. $\texttt{MILP}_{\texttt{MIPVerify}}$— encodes the query in Gurobi using MIPVerify's MILP encoding [63].[6]

We then evaluate three configurations of SoI-based complete analysis parameterized by the branching heuristic and the SoI-minimization algorithm: 1. $\texttt{SOI}^{\texttt{snc}}_{\texttt{mcmc}}$—

[6] This configuration does not use the LP/MILP-based preprocessing passes from MIPVerify [63] because they degrade performance on our benchmarks.

Bench. (#)	$\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$		PeregriNN		$\texttt{ERAN}_1$		$\texttt{ERAN}_2$	
	Solv.	Time	Solv.	Time	Solv.	Time	Solv.	Time
MNIST$_1$ (90)	70	5976	64	11117	**76**	18679	75	19520
MNIST$_2$ (90)	**31**	757	31	2287	28	1910	28	3126
MNIST$_3$ (90)	**35**	8327	26	2344	24	1538	24	3292
Taxi$_1$ (90)	89	1390	-	-	**90**	1653	90	3262
Taxi$_2$ (90)	**71**	6889	-	-	40	16460	35	31778
Taxi$_3$ (90)	88	2164	-	-	**88**	1389	88	4581
CIFAR10$_\text{b}$ (90)	73	16469	-	-	**77**	4604	77	14269
CIFAR10$_\text{w}$ (90)	**42**	8129	-	-	41	14403	37	14453
CIFAR10$_\text{d}$ (90)	**51**	15056	-	-	31	7587	26	5245
GTSR$_1$ (90)	90	15238	-	-	**90**	2023	90	32585
GTSR$_2$ (90)	**90**	10426	-	-	78	77829	75	81232
Total (990)	**730**	90822	-	-	663	148075	645	213343

Table 2: Instances solved by different complete verifiers and their runtime (in seconds) on solved instances.

runs `DeepSoI` with the SnC branching heuristic; 2. $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$— runs `DeepSoI` with the pseudo-impact (PI) heuristic; 3. $\texttt{SOI}^{\texttt{pi}}_{\texttt{wsat}}$— runs the Walksat-based algorithm with the PI heuristic. Each SoI configuration differs in one parameter w.r.t. the previous, so that pair-wise comparison highlights the effect of that parameter.

Table 1 summarizes the runtime performance of different configurations on the four benchmark sets. The three configurations that minimize the SoI, namely $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$, $\texttt{SOI}^{\texttt{pi}}_{\texttt{wsat}}$ and $\texttt{SOI}^{\texttt{snc}}_{\texttt{mcmc}}$, all solve significantly more instances than the two baseline configurations. In particular, $\texttt{SOI}^{\texttt{snc}}_{\texttt{mcmc}}$ solves 248 (53.4%) more instances than $\texttt{LP}^{\texttt{snc}}$. Since all configurations start with the same variable bounds derived by the DeepPoly analysis, the performance gain is mainly due to the use of SoI.

Among the three SoI configurations, the one with both `pi` and `mcmc` solves the most instances. In particular, it solves 8 more instances than $\texttt{SOI}^{\texttt{pi}}_{\texttt{wsat}}$, suggesting that MCMC sampling is, overall, a better approach than the Walksat-based strategy. On the other hand, $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$ and $\texttt{SOI}^{\texttt{snc}}_{\texttt{mcmc}}$ show complementary behaviors. For instance, the latter solves 5 more instances on $\texttt{MNIST}_1$, and the former solves 11 more on the `Taxi` benchmarks. This motivates a portfolio configuration $\texttt{SOI}_{\texttt{portfolio}}$,which runs $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$ and $\texttt{SOI}^{\texttt{snc}}_{\texttt{mcmc}}$ in parallel. This strategy is able to solve 742 instances overall with a 1-hour wall-clock timeout, yielding a gain of at least 12 more solved instances compared with any single-threaded configuration.

6.5 Comparison with other complete analyzers.

In this section, we compare our implementation with other complete analyzers. We first compare with PeregriNN, which as described in Section 2 introduces a heuristic cost function to guide the search. We evaluate PeregriNN on the `MNIST` networks, the same set of networks used in its original evaluation. We did not run PeregriNN on the other benchmarks because it only supports `.nnet` format, which is designed for fully connected feed-forward ReLU networks.

In addition, we also compare with ERAN, a state-of-the-art complete analyzer based on abstract interpretation, on the full set of benchmarks. ERAN is often used as a strong baseline in recent neural network verification literature and was among the top performers in the past VNN Competition 2021. We compare with two ERAN configurations: 1. $\texttt{ERAN}_1$ — ERAN using the DeepPoly analysis [58] for abstract interpretation and Gurobi for solving; 2. $\texttt{ERAN}_2$ — same as above except using the k-ReLU analysis [56] for abstract interpretation. We choose to compare with ERAN instead of other state-of-the-art neural network analyzers, e.g., alpha-beta crown [76,68], OVAL [19], and fast-and-complete [75], mainly because the latter tools are GPU-based, while ERAN supports execution on CPU, where our prototype is designed to run. This makes a fair comparison possible. Note that our goal in this section is not to claim superiority over all state-of-the-art solvers. Rather, the goal is to provide assurance that our implementation is reasonable. As explained earlier, our approach can be integrated into other complete search shells with different search heuristics, and is orthogonal to techniques such as GPU-acceleration, parallelization, and tighter convex relaxation (e.g., beyond the Planet relaxation), which are all future development directions for Marabou.

Table 2 summarizes the runtime performance of different solvers. We include again our best configuration, $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$, for ease of comparison. On the three MNIST benchmark sets, PeregriNN either solves fewer instances than $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$ or takes longer time to solve the same number of instances. We note that PeregriNN's heuristic objective function could be employed during the feasibility check of DeepSoI (Line 4, Algorithm 2). Exploring this complementarity between PeregriNN and our approach is left as future work.

Compared with $\texttt{ERAN}_1$ and $\texttt{ERAN}_2$, $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$ also solves significantly more instances overall, with a performance gain of at least 10.1% more solved instances. Taking a closer look at the performance breakdown on individual benchmarks, we observe complementary behaviors between $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$ and $\texttt{ERAN}_1$, with the latter solving more instances than $\texttt{SOI}^{\texttt{pi}}_{\texttt{mcmc}}$ on 3 of the 11 benchmark sets. Figure 3 shows the cactus plot of configurations that run on all benchmarks. $\texttt{ERAN}_1$ is able to solve more instances than all the other configurations when the time limit is short, but is overtaken by the three SoI-based configurations once the time limit exceeds 30s. One explanation for this is that the SoI-enabled configurations spend more time probing at each search state, and for easier instances, it might be more beneficial to branch eagerly.

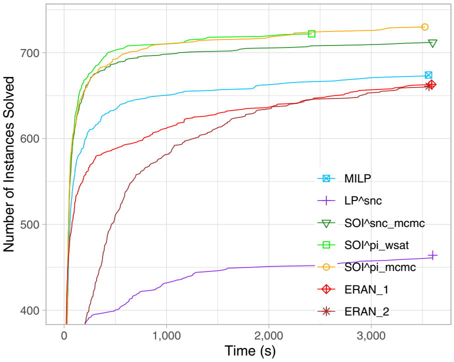


Fig. 3: Cactus plot on all benchmarks.

Finally, we compare the portfolio strategy $\texttt{SOI}_{\texttt{portfolio}}$ described in the previous subsection to $\texttt{ERAN}_1$ running 2 threads. The latter solves 10.3% fewer instances (673 overall). Figure 4 shows a scatter plot of the runtime performance of these two configurations. For unsatisfiable instances, most can be resolved efficiently by both solvers, and each solver has a few unique solves. On the other hand, $\texttt{SOI}_{\texttt{portfolio}}$ is able to solve significantly more satisfiable benchmarks.

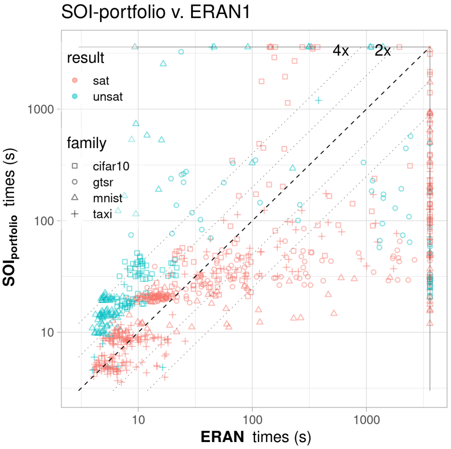


Fig. 4: Runtime of $\texttt{SOI}_{\texttt{portfolio}}$ and $\texttt{ERAN}_1$ running with 2 threads.

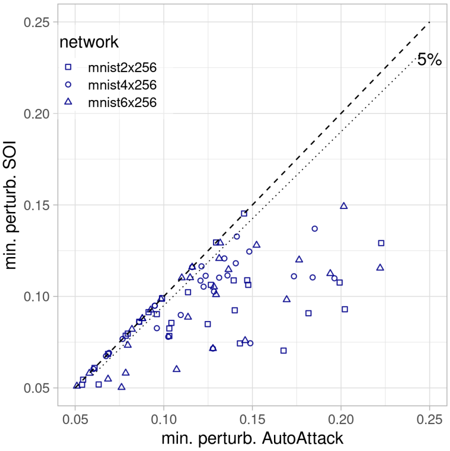


Fig. 5: Improvements over the perturbation bounds found by AutoAttack.

6.6 Incremental Solving and the Rejection Threshold T

The rejection threshold T in Algorithm 2 controls the number of rejected proposals allowed before returning UNKNOWN. An *incremental* solver is one that can accept a sequence of queries, accumulating and reusing relevant bounds derived by each query. To investigate the interplay of T and incrementality, we perform an experiment using the incremental simplex engine in Marabou while varying the value of T. We additionally control the branching order (by using a fixed topological order). We conduct the experiment on 180 $\texttt{MNIST}_1$ and 180 $\texttt{Taxi}_1$ benchmarks from the aforementioned distributions.

Table 3 shows the number of solved instances, as well as the average time (in seconds) and number of search states on the 95 commonly solved UNSAT instances. As T increases, more satisfiable benchmarks are solved.

Rejection threshold T	1	2	3	4	5	6
SAT Solv.	192	199	196	204	203	**207**
UNSAT Solv.	**91**	90	90	89	90	89
Avg. time (common)	97.75	129.0	**83.6**	108.1	137.0	187.8
Avg. states (common)	12948	12712	6122	**5586**	6404	8948

Table 3: Effect of the rejection threshold.

Increasing T can also result in improvement on unsatisfiable instances—either the average time decreases, or fewer search states are required to solve the same instance. We believe this improvement is due to the reuse of bounds derived during the execution of DeepSoI. This suggests that adding incrementality to the convex solver (like Gurobi) could be highly beneficial for verification applications. It also suggests that the bounds derived during the simplex execution cannot be subsumed by bound-tightening analyses such as DeepPoly.

6.7 Improving the perturbation bounds found by AutoAttack

Our proposed techniques result in significant performance gain on satisfiable instances. It is natural to ask whether the satisfiable instances solvable by the SoI-enabled analysis can also be easily handled by adversarial attack algorithms, which as mentioned in Section 2, focus solely on finding satisfying assignments. In this section, we show that this is not the case by presenting an experiment where we use our procedure in combination with AutoAttack [17], a state-of-the-art adversarial attack algorithm, to find higher-quality adversarial examples.

Conceretely, we first use AutoAttack to find an upper bound on the minimal perturbation required for a successful l_∞ attack.We then use our procedure to search for smaller perturbation bounds, repeatedly decreasing the bound by 2% until either UNSAT is proven or a timeout (30 minutes) is reached. We use the adversarial label of the last successful attack found by AutoAttack as the target label. We do this for the first 40 correctly classified test images for the three MNIST architectures, which yields 120 instances. Figure 5 shows the improvement of the perturbation bounds. Reduction of the bound is obtained for 53.3% of the instances, with an average reduction of 26.3%, a median reduction of 22%, and a maximum reduction of 58%. This suggests that our procedure can help obtain a more precise robustness estimation.

7 Conclusions and Future Work

In this paper, we introduced a procedure, `DeepSoI`, for efficiently minimizing the sum of infeasibilities in activation function constraints with respect to the convex relaxation of a neural network verification query. We showed how `DeepSoI` can be integrated into a complete verification procedure, and we introduced a novel SoI-enabled branching heuristic. Extensive experimental results suggest that our approach is a useful contribution towards scalable analysis of neural networks. Our work also opens up multiple promising future directions, including: 1) improving the scalability of `DeepSoI` by using heuristically chosen subsets of activation functions in the cost function instead of all unfixed activation functions; 2) leveraging parallelism by using GPU-friendly convex procedures or minimizing the SoI in a distributed manner; 3) devising more sophisticated initialization and proposal strategies for the Metropolis-Hastings algorithm; 4) understanding the effects of the proposed branching heuristics on different types of benchmarks; 5) investigating the use of `DeepSoI` as a stand-alone adversarial attack algorithm.

Acknowledgements We thank Gagandeep Singh for providing useful feedback and Haitham Khedr for help with running PeregriNN. This work was partially supported by DARPA (grant FA8750-18-C-0099), a Ford Alliance Project (199909), NSF (grant 1814369), and the US-Israel Binational Science Foundation (grant 2020250).

References

1. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: A synergistic approach for analyzing neural network robustness. In: Proc. Programming Language Design and Implementation (PLDI). p. 731–744 (2019)
2. Anderson, R., Huchette, J., Ma, W., Tjandraatmadja, C., Vielma, J.P.: Strong mixed-integer programming formulations for trained neural networks. Mathematical Programming pp. 1–37 (2020)
3. Andrieu, C., De Freitas, N., Doucet, A., Jordan, M.I.: An introduction to mcmc for machine learning. Machine learning **50**(1), 5–43 (2003)
4. Bak, S., Liu, C., Johnson, T.: The second international verification of neural networks competition (vnn-comp 2021): Summary and results. arXiv preprint arXiv:2109.00498 (2021)
5. Bak, S., Tran, H.D., Hobbs, K., Johnson, T.T.: Improved geometric path enumeration for verifying relu neural networks. In: International Conference on Computer Aided Verification. pp. 66–96. Springer (2020)
6. Bénichou, M., Gauthier, J.M., Girodet, P., Hentges, G., Ribière, G., Vincent, O.: Experiments in mixed-integer linear programming. Mathematical Programming **1**(1), 76–94 (1971)
7. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., et al.: End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316 (2016)
8. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 3240–3247 (2019)
9. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of relu-based neural networks via dependency analysis. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 3291–3299 (2020)
10. Boyd, S., Boyd, S.P., Vandenberghe, L.: Convex optimization. Cambridge university press (2004)
11. Brooks, S., Gelman, A., Jones, G., Meng, X.L.: Handbook of markov chain monte carlo. CRC press (2011)
12. Bunel, R., De Palma, A., Desmaison, A., Dvijotham, K., Kohli, P., Torr, P., Kumar, M.P.: Lagrangian decomposition for neural network verification. In: Conference on Uncertainty in Artificial Intelligence. pp. 370–379. PMLR (2020)
13. Bunel, R., Lu, J., Turkaslan, I., Kohli, P., Torr, P., Mudigonda, P.: Branch and bound for piecewise linear neural network verification. Journal of Machine Learning Research **21**(2020) (2020)
14. Bunel, R.R., Turkaslan, I., Torr, P., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 31. Curran Associates, Inc. (2018), https://proceedings.neurips.cc/paper/2018/file/be53d253d6bc3258a8160556dda3e9b2-Paper.pdf
15. Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: 2017 ieee symposium on security and privacy (sp). pp. 39–57. IEEE (2017)
16. Chib, S., Greenberg, E.: Understanding the metropolis-hastings algorithm. The american statistician **49**(4), 327–335 (1995)
17. Croce, F., Hein, M.: Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In: International conference on machine learning. pp. 2206–2216. PMLR (2020)

18. Dantzig, G.B., Orden, A., Wolfe, P., et al.: The generalized simplex method for minimizing a linear form under linear inequality restraints. Pacific Journal of Mathematics **5**(2), 183–195 (1955)
19. De Palma, A., Behl, H., Bunel, R.R., Torr, P., Kumar, M.P.: Scaling the convex barrier with active sets. In: International Conference on Learning Representations (2020)
20. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings (2018)
21. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: UAI. vol. 1, p. 3 (2018)
22. Dvijotham, K.D., Stanforth, R., Gowal, S., Qin, C., De, S., Kohli, P.: Efficient neural network verification with exactness characterization. In: Uncertainty in Artificial Intelligence. pp. 497–507. PMLR (2020)
23. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: International Symposium on Automated Technology for Verification and Analysis. pp. 269–286. Springer (2017)
24. Fischetti, M., Jo, J.: Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. CoRR **abs/1712.06174** (2017)
25. Fromherz, A., Leino, K., Fredrikson, M., Parno, B., Păsăreanu, C.: Fast geometric projections for local robustness certification. arXiv preprint arXiv:2002.04742 (2020)
26. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. pp. 3–18 (2018). https://doi.org/10.1109/SP.2018.00058, https://doi.org/10.1109/SP.2018.00058
27. Gent, I.P., IRST, T.: Hybrid problems, hybrid solutions 73 j. hallam et al.(eds.) ios press, 1995 unsatisfied variables in local search. Hybrid problems, hybrid solutions **27**, 73 (1995)
28. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
29. Henriksen, P., Lomuscio, A.: Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis. In: Zhou, Z.H. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21. pp. 2549–2555. International Joint Conferences on Artificial Intelligence Organization (8 2021). https://doi.org/10.24963/ijcai.2021/351, https://doi.org/10.24963/ijcai.2021/351, main Track
30. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal processing magazine **29**(6), 82–97 (2012)
31. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV (2017)
32. Jia, Z., Zaharia, M., Aiken, A.: Beyond data and model parallelism for deep neural networks. In: Talwalkar, A., Smith, V., Zaharia, M. (eds.) Proceedings of Machine Learning and Systems. vol. 1, pp. 1–13 (2019), https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf
33. Julian, K.D., Lee, R., Kochenderfer, M.J.: Validation of image-based neural network controllers through adaptive stress testing. In: 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC). pp. 1–7. IEEE (2020)

34. Kass, R.E., Carlin, B.P., Gelman, A., Neal, R.M.: Markov chain monte carlo in practice: a roundtable discussion. The American Statistician **52**(2), 93–100 (1998)
35. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In: Proc. 29th Int. Conf. on Computer Aided Verification (CAV). pp. 97–117 (2017)
36. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452 (2019)
37. Khedr, H., Ferlez, J., Shoukry, Y.: Peregrinn: Penalized-relaxation greedy neural network verifier. arXiv preprint arXiv:2006.10864 (2020)
38. King, T.: Effective algorithms for the satisfiability of quantifier-free formulas over linear real and integer arithmetic. Ph.D. thesis, Citeseer (2014)
39. King, T., Barrett, C., Dutertre, B.: Simplex with sum of infeasibilities for smt. In: 2013 Formal Methods in Computer-Aided Design. pp. 189–196. IEEE (2013)
40. Krizhevsky, A., Nair, V., Hinton, G.: Cifar-10 (canadian institute for advanced research). URL http://www. cs. toronto. edu/kriz/cifar. html **5**(4), 1 (2010)
41. Kurakin, A., Goodfellow, I.J., Bengio, S.: Adversarial examples in the physical world. In: ICLR (Workshop). OpenReview.net (2017)
42. LeCun, Y., Cortes, C.: MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/ (2010), http://yann.lecun.com/exdb/mnist/
43. Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. arXiv preprint arXiv:1903.06758 (2019)
44. Lu, J., Kumar, M.P.: Neural network branching for neural network verification. arXiv preprint arXiv:1912.01329 (2019)
45. Lyu, Z., Ko, C.Y., Kong, Z., Wong, N., Lin, D., Daniel, L.: Fastened crown: Tightened neural network robustness certificates. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 5037–5044 (2020)
46. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. arXiv preprint arXiv:1706.06083 (2017)
47. Masi, I., Wu, Y., Hassner, T., Natarajan, P.: Deep face recognition: A survey. In: 2018 31st SIBGRAPI conference on graphics, patterns and images (SIBGRAPI). pp. 471–478. IEEE (2018)
48. Müller, C., Serre, F., Singh, G., Püschel, M., Vechev, M.: Scaling polyhedral neural network verification on gpus. Proceedings of Machine Learning and Systems **3** (2021)
49. Müller, M.N., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.: Precise multi-neuron abstractions for neural network certification. arXiv preprint arXiv:2103.03638 (2021)
50. Paterson, C., Wu, H., Grese, J., Calinescu, R., Pasareanu, C.S., Barrett, C.: Deepcert: Verification of contextually relevant robustness for neural network image classifiers. arXiv preprint arXiv:2103.01629 (2021)
51. Raghunathan, A., Steinhardt, J., Liang, P.: Semidefinite relaxations for certifying robustness to adversarial examples. arXiv preprint arXiv:1811.01057 (2018)
52. Salman, H., Yang, G., Zhang, H., Hsieh, C.J., Zhang, P.: A convex relaxation barrier to tight robustness verification of neural networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019), https://proceedings.neurips.cc/paper/2019/file/246a3c5544feb054f3ea718f61adfa16-Paper.pdf

53. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. ACM SIGARCH Computer Architecture News **41**(1), 305–316 (2013)
54. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: AAAI. vol. 94, pp. 337–343 (1994)
55. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484 (2016)
56. Singh, G., Ganvir, R., Püschel, M., Vechev, M.: Beyond the single neuron convex barrier for neural network certification. Advances in Neural Information Processing Systems **32**, 15098–15109 (2019)
57. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. Advances in Neural Information Processing Systems **31**, 10802–10813 (2018)
58. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
59. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (2019)
60. Stallkamp, J., Schlipsing, M., Salmen, J., Igel, C.: The german traffic sign recognition benchmark: a multi-class classification competition. In: The 2011 international joint conference on neural networks. pp. 1453–1460. IEEE (2011)
61. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
62. Tjandraatmadja, C., Anderson, R., Huchette, J., Ma, W., PATEL, K.K., Vielma, J.P.: The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 21675–21686. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper/2020/file/f6c2a0c4b566bc99d596e58638e342b0-Paper.pdf
63. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019), https://openreview.net/forum?id=HyGIdiRqtm
64. Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. In: International Conference on Computer Aided Verification. pp. 18–42. Springer (2020)
65. Vincent, J.A., Schwager, M.: Reachable polyhedral marching (rpm): A safety verification algorithm for robotic systems with deep neural network components. arXiv preprint arXiv:2011.11609 (2020)
66. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 6369–6379 (2018), http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks
67. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1599–1614 (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

68. Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.J., Kolter, J.Z.: Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. arXiv preprint arXiv:2103.06624 (2021)
69. Weng, L., Zhang, H., Chen, H., Song, Z., Hsieh, C.J., Daniel, L., Boning, D., Dhillon, I.: Towards fast computation of certified robustness for relu networks. In: International Conference on Machine Learning. pp. 5276–5285. PMLR (2018)
70. Wong, E., Kolter, Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: International Conference on Machine Learning. pp. 5286–5295. PMLR (2018)
71. Wu, H., Ozdemir, A., Zeljić, A., Julian, K., Irfan, A., Gopinath, D., Fouladi, S., Katz, G., Pasareanu, C., Barrett, C.: Parallelization techniques for verifying neural networks. In: 2020 Formal Methods in Computer Aided Design (FMCAD). pp. 128–137. IEEE (2020)
72. Wu, H., Zeljić, A., Katz, G., Barrett, C.: Artifact for Paper Efficient Neural Network Analysis with Sum-of-Infeasibilities (Feb 2022), https://doi.org/10.5281/zenodo.6109456
73. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE transactions on neural networks and learning systems **29**(11), 5777–5783 (2018)
74. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.W., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.J.: Automatic perturbation analysis for scalable certified robustness and beyond. Advances in Neural Information Processing Systems **33** (2020)
75. Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X., Hsieh, C.J.: Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. arXiv preprint arXiv:2011.13824 (2020)
76. Zhang, H., Weng, T.W., Chen, P.Y., Hsieh, C.J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 31. Curran Associates, Inc. (2018), https://proceedings.neurips.cc/paper/2018/file/d04863f100d59b3eb688a11f95b0ae60-Paper.pdf

Blockchain

Formal Verification of the Ethereum 2.0 Beacon Chain*

Franck Cassez (✉)[1], Joanne Fuller[1], and Aditya Asgaonkar[2]

[1] ConsenSys, New York, USA
[2] Ethereum Foundation, Zug, Switzerland
franck.cassez@consensys.net joanne.fuller@consensys.net
aditya.asgaonkar@ethereum.org

Abstract. We report our experience in the formal verification of the reference implementation of the Beacon Chain. The Beacon Chain is the backbone component of the new Proof-of-Stake Ethereum 2.0 network: it is in charge of tracking information about the *validators*, their *stakes*, their *attestations* (votes) and if some validators are found to be dishonest, to *slash* them (they lose some of their stakes). The Beacon Chain is mission-critical and any bug in it could compromise the whole network. The *Beacon Chain reference implementation* developed by the Ethereum Foundation is written in Python, and provides a detailed operational description of the state machine each Beacon Chain's network participant (node) must implement. We have formally specified and verified the absence of runtime errors in (a large and critical part of) the Beacon Chain reference implementation using the verification-friendly language Dafny. During the course of this work, we have uncovered several issues, proposed *verified* fixes. We have also synthesised *functional correctness specifications* that enable us to provide guarantees beyond runtime errors. Our software artefact with the code and proofs in Dafny is available at https://github.com/ConsenSys/eth2.0-dafny.

1 Introduction

The Ethereum network is gradually transitioning to a more secure, scalable and energy efficient *Proof-of-Stake (PoS) consensus protocol*, known as Ethereum 2.0 and based off GasperFFG [2]. The Proof-of-Stake discipline ensures that participants who propose (and vote) for blocks are chosen with a frequency that is proportional to their stakes. Another major feature of Ethereum 2.0 is *sharding* which enables the main blockchain to split into a number of independent and hopefully smaller and faster chains. The transition from the current Ethereum 1 to the final version of Ethereum 2.0 (Serenity) is planned over a number of years and will be rolled out in a number of phases. The first phase, Phase 0, is known as the Beacon Chain. It is the backbone component of Ethereum 2.0 as it coordinates the whole network of *stakers* and shards.

* This work was partially supported by the Ethereum Foundation, grant FY20-285, Q4-2020.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 167–182, 2022.
https://doi.org/10.1007/978-3-030-99524-9_9

The Beacon Chain. The Beacon Chain (and its underlying protocol) is in charge of enforcing consensus, among the nodes, called *validators*, participating in the network, on the state of the system. The set of validators is dynamic: new validators can register by staking some ETH (Ethereum crypto-currency). Once registered, validators are eligible to participate and *propose* and *vote* for new blocks (of transactions) to be appended to the blockchain. The Beacon Chain shipped on December 1, 2020. At the time of writing (October 14, 2021), close to $250,000$ validators have staked $7,780,000$ ETH ($30 Billion USD). Considering the coordination role and the amount of assets managed by the Beacon Chain, it is a mission-critical component of the Ethereum 2.0 ecosystem. The *Beacon Chain reference implementation* developed by the Ethereum Foundation is written in Python, and provides a detailed operational description of the state machine each Beacon Chain's network participant (node) must implement.

Our Contribution. Our contribution is many-fold:

- We have formally specified and verified the absence of runtime errors in (a large and critical part of) the Beacon Chain reference implementation using the verification-friendly language Dafny.
- During the course of this work, we have uncovered several issues, proposed *verified* fixes, some of which have been integrated in the reference implementation, and others have resulted in sunstnatial improvements (accuracy, readability) of the reference implementation.
- We have also manually synthesised *functional correctness specifications* that enable us to provide guarantees beyond runtime errors.
- Our software artefact with the code and proofs in Dafny is publicly available in our repository at https://github.com/ConsenSys/eth2.0-dafny .

Related Work. The Ethereum Foundation has supported several projects related to applying formal methods for the analysis of the Beacon Chain (and other components). A foundational project[3] was undertaken in 2019 by Runtime Verification Inc. and provided a formal and *executable* semantics in the K framework, to the reference implementation [1]. The semantics was validated and the reference implementation could be tested which resulted in a first set of recommendations and fixes to the reference implementation. Although it may be possible to formally verify the Beacon Chain with the K-framework tools, to the best of our knowledge it has not been done yet. Runtime Verification Inc. have also formally specified and verified (in Coq [11]) the underlying GasperFFG [2] protocol. Our work complements these formal verification projects. Indeed, our objective is to provide guarantees for the *absence of bugs* (runtime errors), and *loop termination* which goes beyond testing. We have chosen to use a verification-friendly programming language, Dafny [10], as it enables us to write the code in a more developer-friendly manner (compared to K).

[3] https://github.com/runtimeverification/beacon-chain-spec

2 The Beacon Chain Reference Implementation

In this section we introduce the system we want to formally verify, what are the potential benefits and impacts of such of study, and we set out the goals of our experiment.

2.1 System Description and Scope of the Study

As a robust decentralised system, the Beacon Chain aims to implement a *replicated state machine* [9] that is fault-tolerant to a fraction of unreliable participants (e.g., participants that can crash). The replicated state machine is implemented with a number of networked identical state machines running concurrently. This provides redundancy and a more reliable system. The state of each machine changes on an occurrence of an *event*. As the machines operate asynchronously, two different machines may receive different events that cannot be totally ordered time-wise. This is why before processing an event and changing their states, the state machines run a *consensus protocol* to decide which event they should all process next. The consensus protocol aims to guarantee (under certain conditions) that an agreement will be reached which ensures that events are processed in the same order on each machine.

2.2 The Beacon Chain Reference Implementation

The Beacon Chain (Phase 0) reference implementation [6] describes the *state machine* that every Beacon node (participant) has to implement. The idea is that anyone is allowed to be a participant in the decentralised Ethereum 2.0 ecosystem when it is fully deployed. However, as the consensus protocol is Proof-of-Stake there must be a mechanism for participants to register and stake, to slash a participant's stake if they are caught[4] misbehaving, i.e., not following the consensus protocol, and to reward them if they are honest. The Beacon Chain provides these mechanisms. It maintains records about the participants, called *validators*, ensuring fairness (each honest participant should have a voting power, for new blocks, related to its stake), and safety (a dishonest participant may be slashed and lose part of their stakes).

The full Beacon Chain (Phase 0) reference implementation [6] comprises three main sections:

1. the *Beacon Chain State Transition* describing the Beacon state machine which is the most complex component;
2. The *Simple SerialiZe (SSZ) library* for how to encode/decode (serialise/deserialise) data that have to be communicated over the network;
3. the *Merkleise library* for how to build efficient encoding of data structures into Merkle trees, and how to use them to verify Merkle proofs.

[4] In a distributed system with potentially dishonest participants, it is not always possible to detect who is dishonest (byzantine). However, sometimes a participant can sometimes be proved to be dishonest.

The State Transition. The *Beacon Chain state transition* part is the most critical part and at the operational level the complexity stems from:

- time is logically divided into *epochs*, and each epoch into a fixed number of *slots*; the state is updated at each slot;
- at the beginning of each epoch, disjoint subsets of validators are assigned to each slot to participate in the block proposal for the slot and attest (vote) for *links* in the chain;
- the state updates that apply at an epoch boundary are more complex than the other updates;
- the actual state of the chain is a *block-tree* i.e., a tree of blocks, and the canonical chain is defined as a particular *branch in this tree*. How this branch is determined is defined by the *fork choice rule*.
- the *fork choice* rule relies on properties of nodes, *justification* and *finalisation*, in the block-tree. The state update describes how nodes in the block-tree are deemed justified/finalised. The rules for justification and finalisation are introduced in a separate document, the GasperFFG [2] protocol.

SSZ and Merkleise. These libraries are self-contained and independent from the state transition. We used them as a feasibility study and we had verified them before this project started. We have provided a complete Dafny reference implementation for them in the `merkle` and `ssz` packages [3].

2.3 Motivation for Formal Verification

As mentioned previously, the Beacon Chain shipped on December 1, 2020 and up to date, 250, 000 validators have staked 7, 780, 000 ETH ($30 Billion USD). It is clear that any bug, or logical error, could have disastrous consequences resulting is losses of assets for regular users, or downtimes and degradation of service, or losses of rewards for the validators.

There are regular opportunities (forks) to update the code of Beacon Chain nodes, so continuously running projects like ours is very valuable as what is important is to find and fix bugs before attackers can exploit them. The operational description of the Beacon Chain in the reference implementation is provided in Python. It was written by several reference implementation writers at the Ethereum Foundation and due to its size it is hard for one person to have a complete picture of it. It is the reference for any Beacon Chain client implementer. As a result, inaccuracies, ambiguities, or bugs in the reference implementation will lead to erroneous and/or buggy clients that can compromise the integrity, or the performance of the network. Moreover the reference implementation uses a defensive mechanism against unexpected errors:

> *(Rule 1) "State transitions that trigger an unhandled exception (e.g. a failed assert or an out-of-range list access) are considered invalid. State transitions that cause a uint64 overflow or underflow are also considered invalid." [6]*

However this creates a risk that errors unrelated to the logic of the state transition function may introduce spurious exceptions. At the time of writing, there are at least 4 different Ethereum 2.0 client softwares that are used by validators. Bugs in the reference implementation may be handled differently in the various clients, and in some cases lead to a split in the network[5]. The correctness of the consensus mechanism is guaranteed for up to 1/3 of malicious nodes, that is, nodes deviating from the reference implementation, be it intentionally or unintentionally (e.g., because of a bug in the code). Hence, we should try to make sure we reduce (buggy) unintentionally malicious nodes.

2.4 Objectives of the Study

Our goal is to improve the overall safety, readability and usability of the reference implementation. Testing is of course an option, and Beacon Chain clients all implement some form of testing. In this project we are interested in proving the absence of bugs which goes beyond what testing techniques can do: testing can show the presence of bugs but not their absence (Dijkstra, 1970).

The primary aspect of our project was to make sure that the code was free of runtime errors (e.g., over/underflows, array-out-of-bounds, division-by-zero, . . .). This provides more confidence that when an exception occurs and a state is left unchanged as per (Rule 1), the root cause is a genuine problem related to the state transition having been given an ill-formed block: if `state_transition(state,signed_block)` triggers an exception, it should imply that there is a problem with the `signed_block` not that some intermediate computations resulted in runtime errors. A secondary goal was to try and synthesise *functional specifications* from the reference implementation. This can help developers to design tests, and contributes to the specifications being language-agnostic. For instance, it can help write a client in a functional language which results in a more inclusive ecosystem.

3 Formal Specification and Verification

In this section we present the challenges of the project, motivate our methodology and conclude with our results' breakdown.

3.1 Challenges

The main challenges in this formal verification project are in the verification of the code of the `state_transition` component of the Beacon Chain. The SSZ and Merkleise libraries are much smaller, simpler, and independent components that can be dealt with separately.

The reference implementation for the Beacon Chain [6] introduces data types and algorithms that should be *interpreted* as Python 3 code. As a result it may

[5] A network split can be caused if some clients reject a chain that is being followed by the other clients, which leads to a hard fork-like situation.

not be straightforward for those who are not familiar with Python to understand the meaning of some parts of the code. More importantly, the reference implementation is not executable and may contain type mismatches, incompatible function signatures, and bugs that can result in runtime errors like under-overflows or array-out-of-bounds.

Listing A.1. The state transition function.

```
 1 def state_transition(
 2     state: BeaconState,
 3     signed_block: SignedBeaconBlock,
 4     validate_result: bool=True
 5 ) -> None:
 6 block = signed_block.message
 7 # Process slots (including those with no blocks) since block
 8 process_slots(state, block.slot)
 9 # Verify signature
10 if validate_result:
11     assert verify_block_signature(state, signed_block)
12 # Process block
13 process_block(state, block)
14 # Verify state root
15 if validate_result:
16     assert block.state_root == hash_tree_root(state)
```

A typical function in the reference implementation is written as a sequence of control blocks (including function calls) intertwined with *checks* in the form of `assert` statements. The `state_transition` function (Listing A.1) is the component that computes the update of the Beacon Chain's state. The state (of type `BeaconState`) records some information including the validators' stakes, the subsets of validators (*committees*) allocated to a given slot, and the hashes[6] of the blocks that have already been added to the chain. A state update is triggered when a (signed) *block* is added to Beacon Chain. The state machine implicitly defined by the reference implementation generates sequences of states of the form:

$$s_0 \xrightarrow{b_0} s_1 \xrightarrow{b_1} s_2 \dots \xrightarrow{b_n} s_{n+1} \dots \tag{StateT}$$

where s_0 is given (initial values), b_0 is the *genesis block* and for each $i \geq 1$, $s_{i+1} =$ `state_transition`(s_i, b_i).

There are several challenges in testing or verifying this kind of code:

- the functions calls (lines 8, 13) *mutate* the input variable `state`; those functions also call other functions that mutate the state.
- the semantics is not fully captured by the Python 3 interpretation because of the defensive mechanism [S1] (Section 2.3, page 4).
- a *valid* state transition is the opposite of an *invalid* state transition (characterised by [S1]). Determining when a computation is not going to trigger runtime errors or failed `assert`s is non-trivial. This is due to the use of mutating functions that can contain `assert` statements on values that are the results of intermediate computations.

[6] The actual blocks are recorded in the `Store` which is a separate data structure.

- overall the code in the reference implementation does not explicitly define what *properties* `signed_block` should satisfy to guarantee that executing the function `state_transition(state,signed_block)` is not going to trigger an exception. The implicit semantics of the code is: if an exception occurs in executing `state_transition` with input `signed_block`, then this block must be invalid (assuming `state` is always valid).
  It follows that, if the code contains a bug that triggers a runtime error unrelated to `signed_block` (e.g., an intermediate computation that overflows, or an array-out-of-bounds in a sorting algorithm), `signed_block` is declared invalid and not added to the chain. To alleviate this problem, we have collected the conditions (predicates) under which the addition of a block should not fail, which clearly defines when a block is valid.
- as there is no reference *functional specification* it is not immediate to understand when a block is invalid, and to write (unit) tests.
- finally the correctness of parts of the code rely on hidden assumptions, e.g., the total amount of ETH is X so no overflow should happen.

The challenges pertaining to the SSZ and Merkleise libraries are more manageable. First, the reference implementation is shorter. Second, even if there is no functional specification available, it is reasonably easy to synthesise them. Due to the previous weaknesses, the reference implementation [6] has been the subject of several informal explainers [15,5,6].

3.2 Methodologies

Resource Constraints. Resource-wise, the timeframe for our project was approximately 8 months (October 2020 to June 2021), with a team of two formal verification researchers (first two co-authors) and one Beacon Chain expert researcher (third co-author).

Verification Technique. The reference implementation is **not** the operational description of a distributed system, but rather a sequential state machine, as per (StateT), Section 3.1. Thus, techniques and tools that are adequate for the goals we set are related to *program formal verification*.

There are several techniques to approach program verification, ranging from fully automated (e.g., static analysis/abstract interpretation [4], software model-checking [8]) to interactive theorem proving [13]. Most static analysers are unsound (they cannot prove the absence of bugs) which disqualifies them for our project. It is anticipated that fully automated verification techniques can be effective to detect runtime errors but may have limited applicability to proving functional correctness.

On the other side of the spectrum, interactive theorem provers offer a complete arsenal of logics/rules that can certainly be used for this kind of projects. However they usually require encoding the software to be verified in a high-level mathematical language that is rather different to a language like Python. The level of expertise/experience required to properly use these tools is also high. Overall this seemed incompatible with our available resources.

A middle-ground between fully automated and interactive techniques is deductive verification available in verification-friendly programming languages like Dafny [10], Why3 [7], Viper [12] or Whiley [14]. Deductive verification lets verification engineers *propose* proofs and check them fully automatically.

We opted for Dafny [10], an award-winning verification-friendly language. Dafny is actively maintained[7] and under continuous improvement. It offers imperative/object oriented and functional programming styles. Moreover, some of us had a previous exposure to Dafny (working on the SSZ/Merkleise libraries early in 2020), and we could be fully operational quickly, and it was compatible with our resources. We are convinced that similar results could be achieved with Why3, Viper or Whiley but did not have the resources to launch concurrent experiments.

Verification Strategy. Our strategy to write the Beacon Chain reference implementation in Dafny and detect/fix runtime errors, and prove some functional properties is three-fold:

1. **Identify simplifications.** The reference implementation is complex and trying to encode it fully in Dafny may result in inessential details hindering our verification progress. One example is the different data types (classes) for `Attestations`. There are several variations of the type `Attestations` and functions to convert between them. For our verification purposes, using `PendingAttestations` instead of the fully fledged `Attestations` was adequate. Another example is the abstraction of *hashing* functions. We assumed an *uninterpreted* collision-free hash function as we did not aim to prove any probabilistic properties involving this function.
2. **Translate the reference implementation in Dafny.** This helped the formal verification researchers to familiarise themselves with the reference implementation. During this phase, we focussed on adding *pre* and *post* conditions to the functions of the reference implementation to guarantee the absence of runtime errors. We were also able to prove some interesting invariants: the data structure that contains the block-tree is indeed a *well-formed tree*. This structure is implemented with links from nodes to their parent (where `null` is a possible parent in the code). The invariant states that the block-tree that is built with the `state_transition` function satisfies: *i*) the set of ancestors of any block contain blocks with strictly smaller *slot* number and is finite (no cycles) *ii*) the set of ancestors of any block in the block-tree always contains the genesis block (with slot 0).
3. **Synthesise functional specifications.** In the last phase, we manually synthesised functional specifications for each function in the reference implementation. We proved that each function in the reference implementation satisfied its functional specification. This enabled us to prove more complex properties as we could do the formal reasoning and proofs on the functional specifications and the results would carry over to the reference implementation. This was an effective solution to be able to prove properties of the

[7] https://github.com/dafny-lang/dafny

reference implementation with lots of *mutations* (side-effects) without having to embed them deep in the proofs.

3.3 Results

The complete code base is freely available in [3]. There are several resources apart from the verified code: a Docker container to batch verify the code, and some notes/videos to help navigate the Dafny specifications.

Coverage. We estimated that we have verified 85% of the reference implementation. The remaining 15% are simplifications e.g., data types, or using a fixed set of validators instead of a dynamic set. Adding the remaining details to the released version would require a substantial amount of work and at the same time it seems that the likelihood of finding new issues is low. Since the Beacon Chain has shipped in December 1, 2020, only a few minor issues have been uncovered and promptly fixed which seems to confirm the previous claim.

Absence of Runtime Errors. All of the functions we have implemented in Dafny are annotated with pre (`requires`) and post (`ensures`) conditions that are verified, including *loop termination*. The Dafny version of function `state_transition` is given in Listing A.2. Other functions are written similarly e.g., `process_slots` and `process_block`. The Dafny verifier enforces the absence of runtime errors like division by zero, under/overflows, array-out-of-bounds. It follows that our code base is provably free of this kind of defect. Moreover, additional checks can be added like the `assert` statement at line 28. We have added all the `assert` statements from the reference implementation and proved that they could not be violated. This requires adding suitable preconditions.

Regarding loop termination proofs, most of the proofs are based on relatively simple ranking functions. An example of a non-trivial proof termination can be found in a functional correctness proof: *the ancestors of a given block form a strictly decreasing sequence, slot-wise, and consequently end up in the genesis block.* The corresponding code is in the Forkchoice.dfy file.

Functional Correctness. Beyond the absence of runtime errors, we have synthesised *functional specifications* based off the reference implementation code. For instance we have decomposed the state update in `state_transition` into a sequence of simpler steps, `updateBlock`, `forwardStateToSlot`, `nextSlot` and proved that the result is a composition of these functions. This provides more confidence that the code is functionally correct as our decomposition specifies smaller changes in the state. It also enables us to prove properties on the functional specifications and transfer them to the imperative version of the code.

Impact of our Project. During the course of this projects we have reported several issues, some of them bugs (3), some of them need for clarifications (5) in the reference implementation. The issues we have uncovered are tracked in the *issues tracker* of our github repository. Some of the bugs we reported have been fixed and our *clarifications* category has led to several improvements in the writing of the reference implementation. Moreover, we have provided a fully documented version of the reference implementation in Dafny. The Dafny code contains clear pre and post conditions that can help developers understand the effect of a function and can be used to write unit tests.

Listing A.2. Dafny version of `state_transition`

```
method state_transition(s:BeaconState,b:BeaconBlock)
                                returns (s': BeaconState)
    //  A valid state to start from
    requires |s.validators| == |s.balances|
    requires is_valid_state_epoch_attestations(s)
    //  b must a block compatible with s
    requires isValidBlock(s, b)
    //  Functional correctness
    ensures s' ==
        updateBlock(forwardStateToSlot(nextSlot(s),b.slot),b)
    //  Other post-conditions
    ...
    ensures s'.slot == b.slot
    ensures s'.latest_block_header.parent_root  ==
        hash_tree_root(
            forwardStateToSlot(nextSlot(s), b.slot)
            .latest_block_header
        )
    ensures |s'.validators| == |s'.balances|
    ...
{
    // Finalise slots before b.slot.
    s' := process_slots(s, b.slot);

    //  Process block and compute the new state.
    s' := process_block(s', b);

    //  Verify state root (from eth2.0 specs)
    assert (b.state_root == hash_tree_root(s'));
}
```

Statistics. Table 1, page 11, provides some insights into the actual code, per file. We have tried to keep the size of each file small and provide optimal modularity in the proofs. The files in the packages fall into one of the three categories: `file.dfy` is the Python-reference implementation translated into Dafny; `file.s.dfy` contains the functional specifications we have synthesised and `file.p.dfy` any additional proofs (Lemmas) that are used in the correctness proofs. It is hard to estimate the *lines of code to lines of proofs ratio* for many reasons: *i*) it is not always possible to locate all the proofs in a separate unit (e.g. a module in Dafny), as this can create circular dependencies.

It follows that counting lines of proofs as lines in the Lemmas is not an accurate measure; *ii*) in some of the proofs, we have, on purpose, provided redundant hints. As a result some proofs can be shortened but this may be at the expense of readability (and verification time). For this project, a conservative (and empirical) *lines of code to lines of proofs ratio* seems to be around 1 to 7.

Table 1. Statistics. A file providing functional specifications. A file providing proofs (lemmas in Dafny). **#LoC** (resp. **#DoC**) is the number of lines of code (resp. documentation), **Lem.** the number of proper lemmas, **Imp.** the number of proved imperative functions with pre/post conditions.

Files	Package	#LoC	Lem.	Imp.	#Doc	$\frac{\#Doc}{\#LoC}$ (%)	Proved
ActiveValidatorBounds.p.dfy	beacon	52	3	0	29	56	3
BeaconChainTypes.dfy	beacon	54	0	0	171	317	0
Helpers.dfy	beacon	1003	9	89	670	67	98
Helpers.p.dfy	beacon	136	13	0	114	84	13
Helpers.s.dfy	beacon	136	9	6	67	49	15
AttestationsTypes.dfy	beacon/attestations	30	0	0	68	227	0
ForkChoice.dfy	beacon/forkchoice	229	3	15	172	75	18
ForkChoiceTypes.dfy	beacon/forkchoice	9	0	0	17	189	0
Crypto.dfy	beacon/helpers	7	0	1	3	43	1
EpochProcessing.dfy	beacon/statetransition	384	0	14	127	33	14
EpochProcessing.s.dfy	beacon/statetransition	398	24	0	336	84	24
ProcessOperations.dfy	beacon/statetransition	361	0	11	119	33	11
ProcessOperations.p.dfy	beacon/statetransition	160	10	0	74	46	10
ProcessOperations.s.dfy	beacon/statetransition	410	12	6	137	33	18
StateTransition.dfy	beacon/statetransition	215	0	8	126	59	8
StateTransition.s.dfy	beacon/statetransition	213	11	1	100	47	12
Validators.dfy	beacon/validators	11	0	0	53	482	0
Merkleise.dfy	merkle	504	9	18	135	27	27
BitListSeDes.dfy	ssz	262	7	3	64	24	10
BitVectorSeDes.dfy	ssz	155	4	3	53	34	7
BoolSeDes.dfy	ssz	22	0	2	3	14	2
BytesAndBits.dfy	ssz	90	7	6	44	49	13
Constants.dfy	ssz	104	0	0	36	35	0
IntSeDes.dfy	ssz	130	2	2	20	15	4
Serialise.dfy	ssz	514	3	5	36	7	8
DafTests.dfy	utils	62	0	4	25	40	4
Eth2Types.dfy	utils	227	1	3	77	34	4
Helpers.dfy	utils	220	11	3	103	47	14
MathHelpers.dfy	utils	293	18	6	105	36	24
NativeTypes.dfy	utils	28	0	0	13	46	0
NonNativeTypes.dfy	utils	8	0	0	6	75	0
SeqHelpers.dfy	utils	69	8	2	58	84	10
SetHelpers.dfy	utils	74	6	0	50	68	6
TOTAL		**6570**	**170**	**208**	**3212**	**49**	**378**

4 Findings and Lessons Learned

During the course of our formal verification effort we found subtle bugs and also proposed some clarifications for the reference implementations. In addition, our work was the opportunity to start some discussions about how to improve the readability of the reference implementation, e.g., by using pre and post conditions rather than `assert` statements. In this section we provide more insights into some of the main issues we reported[8], and also on the practicality of this kind of project.

4.1 Array-out-of-bounds Runtime Error

The function `get_attesting_indices` (Listing A.3) is called from within several important components of the `state_transition` function including the processing of rewards and penalties, justification and finalisation, as well as the processing of attestations (votes).

Listing A.3. Python code for `get_attesting_indices`.

```
1  def get_attesting_indices(
2      state: BeaconState,
3      data: AttestationData,
4      bits: Bitlist[MAX1]
5  ) -> Set[ValidatorIndex]:
6  """
7  Return the set of attesting indices corresponding to
8  ``data`` and ``bits``.
9  """
10 committee=get_beacon_committee(state, data.slot, data.index)
11 return
12     # Collect indices in committee for which bits is set
13     set(index for i, index in enumerate(committee) if bits[i])
```

The last line (13) of `get_attesting_indices` collects the indices in the array `committee` that have a corresponding bit set to true in array `bits` and returns it as a *set* of indices. The length of `bits`, noted |`bits`|, is `MAX1`. Consequently, the following relation must be satisfied to avoid an array-out-of-bounds error: |`committee`| $\leq$ `MAX1`. It follows that to prove[9] the absence of array-out-of-bounds error in Dafny, the specification of `get_attesting_indices` (in Dafny) requires a pre-condition, |`get_beacon_committee`(...)| $\leq$ `MAX1` (line 10). This pre-condition naturally imposes a post-condition for `get_beacon_committee` and trying to prove this post-condition we uncovered a very subtle bug: depending on the number of *active validators* V in `state`:

$\mathbf{V \leq 4,194,304}$: there is no array-out-of-bounds error as we can prove that |`get_beacon_committee`(...)| $\leq$ `MAX1` for *all* values of the input parameters `data.slot` and `data.index`,

[8] https://github.com/ConsenSys/eth2.0-dafny/issues

[9] In Dafny, this check is built-in so you cannot avoid this proof.

$4,194,304 < V < 4,196,352$: there is at least one value of the input parameters `data.slot` and `data.index` for which $|$`get_beacon_committee`$(\dots)| >$ `MAX1`, which results in an array-out-of-bounds, and

$4,196,352 \leq V$: for all input combination of `data.slot` and `data.index`, there is an array-out-of-bounds $|$`get_beacon_committee`$(\dots)| >$ `MAX1`.

This previously undocumented bug was difficult to detect. It required many hours of effort to model the dynamics of the problem; the analysis was quite complex due to the multiple interrelated parameter calculations, as well as the use of floored integer division. The full description and the analysis of this bug has been reported as issue[10] to the reference implementation github repository. The issue was confirmed by the reference implementation writers.

4.2 Beyond Runtime Errors

We have also been able to establish some well-formedness properties of the data structure that represents the *block-tree* built by each node. Each added block has a stamp, the *slot number* and a link to its *parent*. The block-tree is the tree representation of the parent relation. The block-tree should satisfy the following properties:

- Every block b except the genesis block has a parent,
- Every block b with parent p is such that the slot of b is strictly larger than the slot of p,
- the transitive closure of the parent relation produces chains of blocks that are totally ordered using the $<$ relation on slot,
- the smallest element of each chain has slot 0 (and consequently is the genesis block).

We have established these properties in ForkChoice.dfy using a list of invariants on the `Store`.

Another noticeable contribution compared to other approaches (like testing) is that we have proved the termination of all loops. For the majority of the loops, the *ranking function* used to prove termination is rather straightforward. An example of a more complicated (decreasing) ranking function can be found in the proof of a (functional correctness) lemma in ForkChoice.dfy: the proof relies on the slot number of a block's parent being strictly smaller than the slot number of a block itself. The lemma establishes that the graph defined by the parent relation on the blocks in the store, is always well-formed and is a (block-)tree: the list of ancestors of any block in the store is ordered (slot-wise) and the smallest element is the genesis block.

4.3 Finalisation and Justification

During the course of the project we benefited from the guidance of the third co-author who has comprehensive expertise in various aspects of the Beacon Chain,

[10] https://github.com/ethereum/consensus-specs/issues/2500

including the *fork choice* part, and identified the *fork choice* implementation of the reference implementation as a component that needed verification.

The *fork choice rules* are designed to identify a *canonical branch* in the block-tree which in turn defines the *canonical chain*. To achieve this goal, we first assumed a fixed set of validators. Then we built a Dafny proof of the GasperFFG [2] protocol and tried to prove properties about the *justified and finalised blocks* in the block-tree. We could mechanically prove Lemmas 4.11 and 5.1, Theorem 5.2 from [2]. Note that a complete proof in Coq is available in [11] but it does not use the Beacon Chain data structures. We only managed to push these properties up to a certain level on the functional specifications of our code base and not on the actual reference implementation. Doing so would require us to add a substantial amount of details and to modify the structure of several proofs which was not doable in our timeframe. This experimental work is archived in branch `goal1` of the repository. There is a currently ongoing work focussing on this topic: designing the mechanised proofs[11] of the refinement soundness of the state transition function (Phase 0) w.r.t. the GasperFFG protocol.

4.4 Reflection

Verification Effort. The effort for formal verification took 16 person-months. This figure is for the *Beacon Chain State Transition* and does not include the time spent on the SSZ and Merkleise libraries that were completed before this project started. The division of time was primarily between the second and third components of the project. Translation of the reference implementation in Dafny, took approximately 6 person-months[12]. Synthesis of functional specifications (manually), including proofs, took approximately 10 person-months. The time allocation for the identification of simplifications is more difficult to assess. Though some consideration was given initially, this aspect was ongoing, as our understanding of the reference implementation evolved.

Trust Base. The validity of the verification results assumes the correctness of the Dafny specification and the Z3 verifier. Dafny is actively maintained and under continuous improvement. And in the rare instance where Dafny behaves unpredictability, bug reports are responded to in a timely manner. During the course of this project a few bugs were reported. For example it was found that the definition of an inconsistent `const` could lead to unsound verification results and reported as an issue[13] (fixed) to the Dafny language github repository.

Practicality of the Approach. The use of Dafny does not require any specific knowledge beyond standard program verification (Hoare style proofs) and first-order logics. There is ample support (videos, tutorials, books) to help learning how to write Dafny programs and proofs. The main difficulties/challenges in writing and verifying projects of this size with Dafny (and the same holds for

[11] https://github.com/runtimeverification/beacon-chain-verification

[12] This translation includes the proof of absence of runtime errors.

[13] https://github.com/dafny-lang/dafny/issues/922

other verification-friendly automated deductive verifiers) are: **1.** when the verification fails, it requires some experience to interpret the verifier feedback and make some progress, and **2.** the unpredictability (time-wise) of the reasoning engine; this is due to the fact that *verification conditions* that are generated by Dafny are in semi-decidable theories of the underlying SMT-solver (Z3). In our experience, adding a seemingly innocuous line of proof may result in either a surge or a drastic reduction of verification time.

5 Conclusion

Overall this project was a significant undertaking. The complexity of the state transition mechanism, combined with the ambitious project scope, makes this one of the largest formal verification projects to be completed using Dafny. Even with the model simplifications, the Python language is not particularly compatible with the fundamentals that underpin formal verification, which presented continual challenges. Upon reflection: *i*) the project would have benefited from a larger team and *ii*) consideration of the application of formal verification methods earlier, ideally within the design process, would have had a positive impact.

The interest generated from this project provided an opportunity to facilitate Dafny training for the reference implementation writers at the Ethereum Foundation. This training included the translation of code into Dafny, as well as the more advanced topic of proof construction. Participants were able to gain insight into the formal verification process which could provide valuable context when drafting future reference implementations and specifications.

Acknowledgements. We thank the anonymous referees for their constructive feedback which helped improve the initial version of the paper. We thank the reference implementation writers at the Ethereum Foundation for their insightful feedback. This project was supported by an Ethereum Foundation Grant, FY20-285 and we thank Danny Ryan (Ethereum Foundation) and Ben Edgington (ConsenSys) for their help in setting up this project and their support and encouragements. We also thank Roberto Saltini (ConsenSys) for his contribution at the early stage of the project.

References

1. Alturki, M., Bogdanas, D., Hathhorn, C., Park, D., Roşu, G.: An executable K model of ethereum 2.0 beacon chain phase 0 specification. Project Report (2020), https://github.com/runtimeverification/beacon-chain-spec
2. Buterin, V., Hernandez, D., Kamphefner, T., Pham, K., Qiao, Z., Ryan, D., Sin, J., Wang, Y., Zhang, Y.X.: Combining GHOST and casper. CoRR **abs/2003.03052** (2020), https://arxiv.org/abs/2003.03052
3. ConsenSys: Formal verification of the ethereum 2.0 specifications in dafny. (2021), https://github.com/ConsenSys/eth2.0-dafny
4. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
5. Edgington, B.: (2020), https://benjaminion.xyz/eth2-annotated-spec/

6. Ethereum Foundation: Beacon chain specifications (2020), https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md
7. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) 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. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8,
8. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4) (Oct 2009). https://doi.org/10.1145/1592434.1592438,
9. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563,
10. Leino, K.R.M.: Accessible software verification with Dafny. IEEE Softw. **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212,
11. Li, E., Serbanuta, T., Diaconescu, D., Zamfir, V., Rosu, G.: Formalizing correct-by-construction casper in coq. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020. pp. 1–3. IEEE (2020). https://doi.org/10.1109/ICBC48266.2020.9169468,
12. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 104–125. IOS Press (2017). https://doi.org/10.3233/978-1-61499-810-5-104,
13. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer Publishing Company, Incorporated (2014)
14. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with Whiley. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems - 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures. Lecture Notes in Computer Science, vol. 11430, pp. 1–37. Springer (2018). https://doi.org/10.1007/978-3-030-17601-3_1,
15. Ryan, D.: (2020), https://notes.ethereum.org/@djrtwo/Bkn3zpwxB#Phase-0-for-Humans-v0100

Fast and Reliable Formal Verification of Smart Contracts with the Move Prover

David Dill, Wolfgang Grieskamp(✉)[*],
Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong

Novi Research, Meta Platforms, Menlo Park, USA

Abstract. The Move Prover (MVP) is a formal verifier for smart contracts written in the Move programming language. MVP has an expressive specification language, and is fast and reliable enough that it can be run routinely by developers and in integration testing. Besides the simplicity of smart contracts and the Move language, three implementation approaches are responsible for the practicality of MVP: (1) an alias-free memory model, (2) fine-grained invariant checking, and (3) monomorphization. The entirety of the Move code for the Diem blockchain has been extensively specified and can be completely verified by MVP in a few minutes. Changes in the Diem framework must be successfully verified before being integrated into the open source repository on GitHub.

Keywords: Smart contracts · formal verification · Move language · Diem blockchain

1 Introduction

The Move Prover (MVP) is a formal verification tool for smart contracts that intends to be used routinely during code development. The verification finishes fast and predictably, making the experience of running MVP similar to the experience of running compilers, linters, type checkers, and other development tools. Building a fast verifier is non-trivial, and in this paper, we would like to share the most important engineering and architectural decisions that have made this possible.

One factor that makes verification easier is applying it to smart contracts. Smart contracts are easier to verify than conventional software for at least three reasons: 1) they are small in code size, 2) they execute in a well-defined, isolated environment, and 3) their computations are typically sequential, deterministic, and have minimal interactions with the environment (e.g., no explicit I/O operations). At the same time, formal verification is more appealing to the advocates for smart contracts because of the large financial and regulatory risks that smart contracts may entail if misbehaved, as evidenced by large losses that have occurred already [29,19,22].

The other crucial factor to the success of MVP is a tight coupling with the *Move* programming language [26]. Move is developed as part of the Diem blockchain [24] and is designed to be used with formal verification from day one. Move is currently

[*] *Correspondence to*: wgrieskamp@gmail.com

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 183–200, 2022.
https://doi.org/10.1007/978-3-030-99524-9_10

co-evolving with MVP. The language supports specifying pre-, post-, and aborts conditions of functions, as well as invariants over data structures and over the content of the global persistent memory (i.e., the state of the blockchain). One feature that makes verification harder is that quantification can be used freely in specifications.

Despite this specification richness, MVP is capable of verifying the full Move implementation of the Diem blockchain (called the Diem framework [25]) in a few minutes. The framework provides functionality for managing accounts and their interactions, including multiple currencies, account roles, and rules for transactions. It consists of about 8,800 lines of Move code and 6,500 lines of specifications (including comments for both), which shows that the framework is extensively specified. More importantly, *verification is fully automated and runs continuously with unit and integration tests*, which we consider a testament to the practicality of the approach. Running the prover in integration tests requires more than speed: it requires reliability, because tests that work sometimes and fail or time out other times are unacceptable in that context.

MVP is a substantial and evolving piece of software that has been tuned and optimized in many ways. As a result, it is not easy to define exactly what implementation decisions lead to fast and reliable performance. However, we can at least identify three major ideas that resulted in dramatic improvements in speed and reliability since the description of an early prototype of MVP [32]:

- an *alias-free memory model* based on Move's semantics, which are similar to the Rust programming language;
- *fine-grained invariant checking*, which ensures that invariants hold at every state, except when developer explicitly suspends them; and
- monomorphization, which instantiates type parameters in Move's generic structures, functions, and specification properties.

The combined effect of all these improvements transformed a tool that worked, but often exhibited frustrating, sometimes random [12], timeouts on complex and especially on erroneous specifications, to a tool that almost always completes in less than 30 seconds. In addition, there have been many other improvements, including a more expressive specification language, reducing false positives, and error reporting.

The remainder of the paper first introduces the Move language and how MVP is used with it, then discusses the design of MVP and the three main optimizations above. There is also an appendix that describes injection of function specifications.

2 Move and the Prover

Move was developed for the Diem blockchain [24], but its design is not specific to blockchains. A Move execution consists of a sequence of updates evolving a *global persistent memory state*, which we just call the *(global) memory*. Similar to other blockchains, updates are a series of atomic transactions. All runtime errors result in a transaction abort, which does not change the blockchain state except to transfer some currency ("gas") from the account that sent the transaction to pay for cost of executing the transaction.

Fig. 1: Account Example Program

```
module Account {
  struct Account has key {
    balance: u64,
  }

  fun withdraw(account: address, amount: u64) acquires Account {
    let balance = &mut borrow_global_mut<Account>(account).balance;
    assert(*balance >= amount, Errors::limit_exceeded());
    *balance = *balance - amount;
  }

  fun deposit(account: address, amount: u64) acquires Account {
    let balance = &mut borrow_global_mut<Account>(account).balance;
    assert(*balance <= Limits::max_u64() - amount, Errors::limit_exceeded());
    *balance = *balance + amount;
  }

  public(script) fun transfer(from: &signer, to: address, amount: u64)
  acquires Account {
    assert(Signer::address_of(from) != to, Errors::invalid_argument());
    withdraw(Signer::address_of(from), amount);
    deposit(to, amount);
  }
}
```

The global memory is organized as a collection of resources, described by Move structures (data types). A resource in memory is indexed by a pair of a type and an address (for example the address of a user account). For instance, the expression `exists<Coin<USD>>(addr)` will be true if there is a value of type `Coin<USD>` stored at `addr`. As seen in this example, Move uses type generics, and working with generic functions and types is rather idiomatic for Move.

A Move application consists of a set of *transaction scripts*. Each script defines a Move function with input parameters but no output parameters. This function updates the global memory and may emit events. The execution of this function can abort because of an abort instruction or implicitly because of a runtime error such as an out-of-bounds vector index.

Programming in Move In Move, one defines transactions via *script functions* which take a set of parameters. Those functions can call other functions. Script and regular functions are encapsulated in *modules*. Move modules are also the place where structs are defined. An illustration of a Move contract is given in Fig. 1 (for a more complete description see the Move Book [26]). The example is a simple account which holds a balance in the struct `Account`, and offers the script function `transfer` to manipulate this resource. Scripts generally have *signer* arguments, which are tokens which represent an account address that has been authenticated by a cryptographic signature. The `assert` statement in the example causes a Move transaction to abort execution if the condition is not met. Notice that Move, similar as Rust, supports references (as in `&signer`) and mutable references (as in `&mut T`). However, references cannot be part of structs stored in global memory.

Fig. 2: Account Example Specification

```
module Account {
  spec transfer {
    let from_addr = Signer::address_of(from);
    aborts_if from_addr == to;
    aborts_if bal(from_addr) < amount;
    aborts_if bal(to) + amount > Limits::max_u64();
    ensures bal(from_addr) == old(bal(from_addr)) - amount;
    ensures bal(to) == old(bal(to)) + amount;
  }

  spec fun bal(acc: address): u64 {
    global<Account>(acc).balance
  }

  invariant forall acc: address where exists<Account>(acc):
    bal(acc) >= AccountLimits::min_balance();

  invariant update forall acc: address where exists<Account>(acc):
    old(bal(acc)) - bal(acc) <= AccountLimits::max_decrease();
}
```

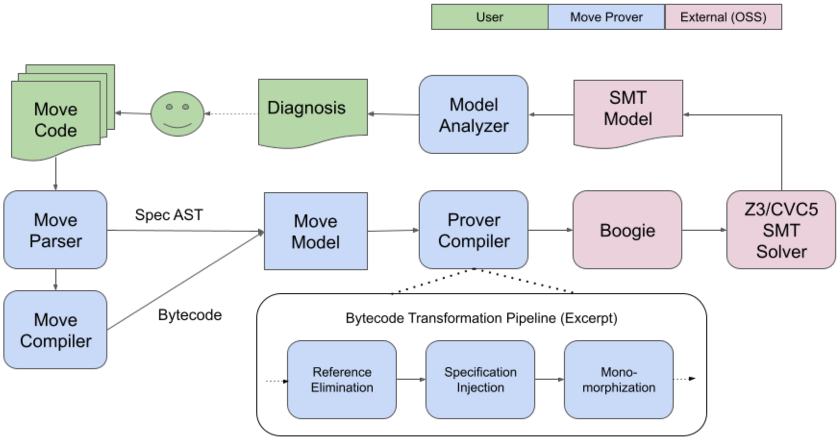
Specifying in Move The specification language supports *Design By Contract* [18]. Developers can provide pre and post conditions for functions, which include conditions over parameters and global memory. Developers can also provide invariants over data structures, as well as the contents of the global memory. Universal and existential quantification over bounded domains, such as like the indices of a vector, as well as effectively unbounded domains, such as memory addresses and integers, are supported. Quantifiers make the verification problem undecidable and cause difficulties with timeouts. However, in practice, we notice that quantifiers have the advantage of allowing more direct formalization of many properties, which increases the clarity of specifications.

Fig. 2 illustrates the specification language by extending the account example in Fig. 1 (for the definition of the specification language see [27]). This adds the specification of the `transfer` function, a helper function `bal` for use in specs, and two global memory invariants. The first invariant states that a balance can never drop underneath a certain minimum. The second invariant refers to an update of global memory with pre and post state: the balance on an account can never decrease in one step more than a certain amount. Note that while the Move programming language has only unsigned integers, the specification language uses arbitrary precision signed integers, making it convenient to specify something like `x + y <= limit`, without the complication of arithmetic overflow.

Specifications for the `withdraw` and `deposit` functions have been omitted in this example. MVP supports omitting specs for non-recursive functions, in which case they are treated as being inlined at caller site.

Running the Prover MVP is fully automatic, like a type checker or linter, and is expected to finish in a reasonable time, so it can be integrated in the regular development workflow. Running MVP on the `module Account` produces multiple errors. The first is this one:

Fig. 3: Move Prover Architecture



```
error: abort not covered by any of the ‘aborts_if‘ clauses
   -- account.move:24:3
   |
13 |       let balance = &mut borrow_global_mut<Account>(account).balance;
   |                          ----------------- abort happened here
   |
   =     at account.move:18: transfer
   =         from = signer{0x18be}
   =         to = 0x18bf
   =         amount = 147u8
   =     at ...
```

MVP detected that an implicit abort condition is missing in the specification of the withdraw function. It prints the context of the error, as well as an *execution trace* which leads to the error. Values of variable assignments from the counterexample found by the SMT solver are printed together with the execution trace. Logically, the counterexample presents an assignment to variables where the program fails to meet the specification. In general, MVP attempts to produce readable diagnostics for Move developers without the need of understanding any internals of the prover.

There are more verification errors in this example, related to the global invariants: the code makes no attempt to respect the limits in min_balance() and max_decrease(). The problem can be fixed by adding more assert statements to check that the limits are met (see full version of the paper [7]).

The programs and specifications MVP deals with are much larger than this example. The conditions under which a transaction in the Diem framework can abort typically involve dozens of individual predicates, stemming from other functions called by this transaction. Moreover, there are hundreds of memory invariants specified, encoding access control and other requirements for the Diem blockchain.

3 Move Prover Design

The architecture of MVP is illustrated in Fig. 3. Move code (containing specifications) is given as input to the tool chain, which produces two artifacts: an abstract syntax tree (AST) of the specifications, and the generated bytecode. The *Move Model* merges both bytecode and specifications, as well as other metadata from the original code, into a unified object model which is input to the remaining tool chain.

The next phase is the actual *Prover Compiler*, which is a pipeline of bytecode transformations. We focus on the transformations shown (Reference Elimination, Specification Injection, and Monomorphization). The Prover uses a modified version of the Move VM bytecode as an intermediate representation for these transformations, but, for clarity, we describe the transformations at the Move source level.

The transformed bytecode is next compiled into the Boogie intermediate verification language [3]. Boogie supports an imperative programming model which is well suited for the encoding of the transformed Move code. Boogie in turn can translate to multiple SMT solver backends, namely Z3 [20] and CVC5 [23]; the default choice for the Move prover is currently Z3.

3.1 Reference Elimination

The reference elimination transformation is what enables the alias-free memory model in the Move Prover, which is one of the most important factors contributing to the speed and reliability of the system. In most software verification and static analysis systems, the explosion in number of possible aliasing relationships between references leads either to high computational complexity or harsh approximations.

In Move, the reference system is based on *borrow semantics* [5] as in the Rust programming language. The initial borrow must come from either a global memory or a local variable on stack (both referred to as *locations* from now on). For local variables, one can create immutable references (with syntax `&x`) and mutable references (with syntax `&mut x`). For global memories, the references can be created via the `borrow_global` and `borrow_global_mut` built-ins. Given a reference to a whole struct, field borrowing can occur via `&mut x.f` and `&x.f`. Similarly, with a reference to a vector, element borrowing occurs via native functions `Vector::borrow(v, i)` and `Vector::borrow_mut(v, i)`. Move provides the following guarantees, which are enforced by the borrow checker:

- For any location, there can be either exactly one mutable reference, or n immutable references. Enforcing this rule is similar to enforcing the borrow semantics in Rust, except for global memories, which do not exist in Rust. For global memories, this rule is enforced via the `acquires` annotations. Using Fig. 1 as an example, function `withdraw` `acquires` the `Account` global location, therefore, `withdraw` is prohibited from calling any other function that might also borrow or modify the `Account` global memory (e.g., `deposit`).
- The lifetime of references to data on the stack cannot exceed the lifetime of the stack location. This includes global memories borrowed inside a function as well—a reference to a global memory cannot be returned from the function, neither in whole nor in parts.

These properties effectively permit the *elimination of references* from a Move program, eliminating need to reason about aliasing.

Immutable References Immutable references are replaced by values. An example of the applied transformation is shown below. We remove the reference type constructor and all reference-taking operations from the code:

```
fun select_f(s: &S): &T { &s.f } ⇝ fun select_f(s: S): T { s.f }
```

When executing a Move program, immutable references are important to avoid copies for performance and to enforce ownership; however, for symbolic reasoning on correct Move programs, the distinction between immutable references and values is unimportant.

Mutable References Each mutation of a location `l` starts with an initial borrow for the whole data stored in this location. This borrow creates a reference `r`. As long as `r` is alive, Move code can either update its value (`*r = v`), or derive a sub-reference (`r' = &mut r.f`). The mutation ends when `r` (and the derived `r'`) go out of scope.

The borrow checker guarantees that during the mutation of the data in `l`, no other reference can exist into the same data in `l` – meaning that it is impossible for other Move code to test whether the value has mutated while the reference is held.

These semantics allow mutable references to be handled via *read-update-write* cycles. One can create a copy of the data in `l` and perform a sequence of mutation steps which are represented as purely functional data updates. Once the last reference for the data in `l` goes out of scope, the updated value is written back to `l`. This converts an imperative program with references into an imperative program which only has state updates on global memory or variables on the stack, with no aliasing. We illustrate the basics of this approach by an example:

```
fun increment(x: &mut u64) { *x = *x + 1 }
fun increment_field(s: &mut S) { increment(&mut s.f) }
fun caller(): S { let s = S{f:0}; update(&mut s); s }
⇝
fun increment(x: u64): u64 { x + 1 }
fun increment_field(s: S): S { s[f = increment(s.f)] }
fun caller(): S { let s = S{f:0}; s = update(s); s }
```

Dynamic Mutable References While the setup in above example covers a majority of the use cases in every day Move code, the general case is more complex, since the referenced location may not be known statically. Consider the following Move code:

```
let r = if (p) &mut s1 else &mut s2;
increment_field(r);
```

Additional information in the logical encoding is required to deal with such cases. When a reference goes out of scope, we need to know from which location it was derived in order to write back the updated value. Fig. 4 illustrates the approach for doing this. Essentially, a new type `Mut<T>`, which is internal to MVP, is introduced to track both the location from which `T` was derived and the value of `T`. `Mut<T>` supports the following operations:

- `Mvp::mklocal(value, LOCAL_ID)` creates a new mutation value for a local with the given local id. A local id uniquely identifies a local variable in the function.

Fig. 4: Elimination of Mutable References

```
fun increment(x: &mut u64) { *x = *x + 1 }
fun increment_field(s: &mut S) {
  let r = if (s.f > 0) &mut s.f else &mut s.g;
  increment(r)
}
fun caller(p: bool): (S, S) {
  let s1 = S{f:0, g:0}; let s2 = S{f:1, g:1};
  let r = if (p) &mut s1 else &mut s2;
  increment_field(r);
  (s1, s2)
}
⇝
fun increment(x: Mut<u64>): Mut<u64> { Mvp::set(x, Mvp::get(x) + 1) }
fun increment_field(s: Mut<S>): Mut<S> {
  let r = if (s.f > 0) Mvp::field(s.f, S_F) else Mvp::field(s.g, S_G);
  r = increment(r);
  if (Mvp::is_field(r, S_F))
    s = Mvp::set(s, Mvp::get(s)[f = Mvp::get(r)]);
  if (Mvp::is_field(r, S_G))
    s = Mvp::set(s, Mvp::get(s)[g = Mvp::get(r)]);
  s
}
fun caller(p: bool): S {
  let s1 = S{f:0, g:0}; let s2 = S{f:1, g:1};
  let r = if (p) Mvp::mklocal(s1, CALLER_s1)
          else Mvp::mklocal(s2, CALLER_s2);
  r = increment_field(r);
  if (Mvp::is_local(r, CALLER_s1))
    s1 = Mvp::get(r);
  if (Mvp::is_local(r, CALLER_s2))
    s2 = Mvp::get(r);
  (s1, s2)
}
```

- Similarly, `Mvp::mkglobal(value, TYPE_ID, addr)` creates a new mutation for a global with given type and address.
- With `r' = Mvp::field(r, FIELD_ID)` a mutation value for a sub-reference is created for the identified field.
- The value of a mutation is replaced with `r' = Mvp::set(r, v)` and retrieved with `v = Mvp::get(r)`.
- With the predicate `Mvp::is_local(r, LOCAL_ID)` one can test whether `r` was derived from the given local, and with `Mvp::is_global(r, TYPE_ID, addr)` for a specific global location. `Mvp::is_field(r, FIELD_ID)` tests whether `r` is derived from the given field.

MVP implements the illustrated transformation by construction a *borrow graph* from the program via data flow analysis. This graph tracks both when references are released as well as how they relate to each other: e.g. `r' = &mut r.f` creates an edge from `r` to `r'` labeled with `f`, and `r' = &mut r.g` creates another also starting from `r`. The borrow analysis is inter-procedural, requiring computed summaries for the borrow graph of called functions.

The resulting borrow graph is then used to guide the transformation, inserting the operations of the `Mut<T>` type as illustrated in Fig 4. Specifically, when the borrow on a reference ends, the associated mutation value must be written back to its parent mutation or the original location (e.g. line 29 in Fig. 4). The presence of multiple possible origins leads to case distinctions via `Mvp::is_X` predicates; however, these cases are rare in actual Move programs.

3.2 Global Invariant Injection

Correctness of smart contracts is largely about the correctness of the blockchain state, so global invariants are particular important in the move specification language. For example, in the Diem framework, global invariants can capture the requirement that an account be accompanied by various other types that are be stored at the same address and the requirement certain state changes are only permitted for certain accounts by the access control scheme.

Most software verification tools prove that functions preserve invariants by assuming the invariant at the entry to each function and proving them at the exit. In a module or class, it is only necessary to prove that invariants are preserved by public functions, since invariants are often violated internally in the implementation of a module or class. An earlier version of the Move Prover used exactly this approach.

The current implementation of the Prover takes the opposite approach: it ensures that invariants hold after every instruction, unless explicitly directed to suspend some invariants by a user. This *fine-grained* approach has performance advantages, because, unless suspended, *invariants are only proven when an instruction is executed that could invalidate them*, and the proofs are often computationally simple because *the change from a single instruction is usually small*. Relatively few invariants are suspended, and, when they are, it is over a relatively small span of instructions, preserving these advantages. There is another important advantage, which is that invariants hold almost everywhere in the code, so they are available to approve other properties, such as abort conditions. For example, if a function accesses type `T1` and then type `T2`, the access to `T2` will never abort if the presence of `T1` implies the presence of `T2` at every state in the body of the function. This situation occurs with some frequency in the Diem framework.

Invariant Types and Proof Methodology *Inductive* invariants are properties declared in Move modules that must (by default) hold for the global memory at all times. Those invariants often quantify over addresses (See Fig. 2 for example.) Based on Move's borrow semantics, inductive invariants don't need to hold while memory is mutated because the changes are not visible to other code until the change is written back. This is reflected by the reference elimination described in Sec. 3.1,

Update invariants are properties that relate two states, a previous state and the current state. Typically they are enforced after an update of global memory. The `old` operator is used to evaluate specification expressions in the previous state.

Verification of both kinds of invariants can be *suspended*. That means, instead of being verified at the time a memory update happens, they are verified at the call site of the function which updates memory. This feature is necessitated by fine-grained invariant checking, because invariants sometimes do not hold in the midst of internal computations of a module. For example, a relationship between state variables may

Fig. 5: Basic Global Invariant Injection

```
fun f(a: address) {
  let r = borrow_global_mut<S>(a);
  r.value = r.value + 1
}
invariant [I1] forall a: address: global<S>(a).value > 0;
invariant [I2] update forall a: address:
    global<S>(a).value > old(global<S>(a).value);
⇝
fun f(a: address) {
  spec assume I1;
  Mvp::snapshot_state(I2_BEFORE);
  r = <increment mutation>;
  spec assert I1;
  spec assert I2[old = I2_BEFORE];
}
```

not hold when the variables are being updated sequentially. Functions with external callers (public or script functions) cannot suspend invariant verification, since the invariants are assumed to hold at the beginning and end of each such function.

Inductive invariants are proven by induction over the evolution of the global memory. The base case is that the invariant must hold in the empty state that precedes the genesis transaction. For the induction step, we can assume that the invariant holds at each verified function entry point for which it is not suspended, and now must prove that it holds after program points which are either direct updates of global memory, or calls to functions which suspend invariants.

For update invariants, no induction proof is needed, since they just relate two memories. The pre-state is some memory captured before an update happens, and the post state the current state.

Modular Verification We wish to support open systems to which untrusted modules can be added with no chance of violating invariants that have already been proven. For each invariant, there is a defined subset of Move modules (called a *cluster*). If the invariant is proven for the modules in the cluster, it is guaranteed to hold in all other modules – even those that were not yet defined when the invariant was proven. The cluster must contain every function that can invalidate the invariant, and, in case of invariant suspension, all callers of such a function. Importantly, functions outside the cluster can never invalidate an invariant. Those functions trivially preserve the invariant, so it is only necessary to verify functions defined in the cluster.

MVP verifies a given set of modules at a time (typically one). The modules being verified are called the *target modules*, and the global invariants to be verified are called *target invariants*, which are all invariants defined in the target modules. The cluster is then the smallest set as specified above such that all target modules are contained.

Basic Translation We first look at injection of global invariants in the absence of type parameters. Fig. 5 contains an example for the supported invariant types and their injection into code. The first invariant, `I1`, is an inductive invariant. It is assumed on function entry, and asserted after the state update. The second, `I2`, is an

Fig. 6: Global Invariant Injection and Genericity

```
invariant [I1] global<S<u64>>(0).value > 1;
invariant<T> [I2] global<S<T>>(0).value > 0;
fun f(a: address) { borrow_global_mut<S<u8>>(0).value = 2 }
fun g<R>(a: address) { borrow_global_mut<S<R>>(0).value = 3 }
⇝
fun f(a: address) {
  spec assume I2[T = u8];
  <<mutate>>
  spec assert I2[T = u8];
}
fun g<R>(a: address) {
  spec assume I1; spec assume I2[T = R];
  <<mutate>>
  spec assert I1; spec assert I2[T = R];
}
```

update invariant, which relates pre and post states. For this a state snapshot is stored under some label I2_BEFORE, which is then used in an assertion.

Global invariant injection is optimized by knowledge of the prover, obtained by static analysis, about accessed and modified memory. Let accessed(f) be the memory accessed by a function, and modified(f) be the memory modified. Let accessed(I) by an invariant (including transitively by all functions it calls).

- Inject assume I at entry to f *if* accessed(f) has overlap with accessed(I).
- Inject assert I after each program step if one of the following is true (a) the step modifies a memory location M in accessed(I) or, (b) the step is a call to function f' in which I is suspended and modifies(f') intersects with accessed (I). Also, if I is an update invariant, inject a save of a memory snaptshot before the update or call.

Genericity Generic type parameters make the problem of determining whether a function can modify an invariant more difficult. Consider the example in Fig. 6. Invariant I1 holds for a specific type instantiation S<u64>, whereas I2 is generic over all type instantiations for S<T>.

The non-generic function f which works on the instantiation S<u8> will have to inject the *specialized* instance I2[T = u8]. The invariant I1, however, does not apply for this function, because there is no overlap with S<u64>. In contrast, g is generic in type R, which could be instantiated to u64. So, I1, which applies to S<u64> needs to be injected in addition to I2.

The general solution depends on type unification. Given the accessed memory of a function f<R> and an invariant I<T>, we compute the pairwise unification of memory types. Those types are parameterized over R resp. T. Successful unification results in a substitution for both type parameters, and we include the invariant with T specialized according to the substitution.

3.3 Monomorphization

Monomorphization is a transformation which removes generic types from a Move program by *specializing* the program for relevant type instantiations. In the context

Fig. 7: Basic Monomorphization

```
struct S<T> { .. }
fun f<T>(x: T) { g<S<T>>(S(x)) }
fun g<S:key>(s: S) { move_to<S>(.., s) }
⇝
struct T{}
struct S_T{ .. }
fun f_T(x: T) { g_S_T(S_T(x)) }
fun g_S_T(s: S_T) { move_to<S_T>(.., s) }
```

of verification, the goal is that the specialized program verifies if and only if the generic program verifies in an encoding which supports types as first class values. We expect the specialized program to verify faster because it avoids the problem of generic representation of values, supporting a multi-sorted representation in the SMT logic.

To verify a generic function for all possible instantiations, monomorphization skolemizes the type parameter, i.e. the function is verified for a new type with no special properties that represents an arbitrary type. It then specializes all called functions and used data types with this new type and any other concrete types they may use. Fig. 7 sketches this approach.

However, this approach has one issue: the type of genericity Move provides does not allow for full type erasure (unlike many programming languages) because types are used to *index* global memory (e.g. `global<S<T>>(addr)` where `T` is a generic type). Consider the following Move function:

```
fun f<T>(..) { move_to<S<T>>(s, ..); move_to<S<u64>>(s, ..) }
```

Depending on how `T` is instantiated, this function behaves differently. Specifically, if `T` is instantiated with `u64` the function will always abort at the second `move_to`, since the target location is already occupied.

The important property enabling monomorphization in the presence of such type dependent code is that one can identify the situation by looking at the memory accessed by code and injected specifications. From this one can derive *additional instantiations of the function* which need to be verified. In the example above, verifying both `f_T` and an instantiation `f_u64` will cover all relevant cases of the function behavior.

The algorithm for computing the instances that require verification works as follows. Let `f<T1,..,Tn>` be a verified target function which has all specifications injected and inlined function calls expanded.

- For each memory `M in modified(f)`, if there is a memory `M' in modified(f) + accessed(f)` such that `M` and `M'` can unify via `T1,..,Tn`, collect an instantiation of the type parameters `Ti` from the resulting substitution. This instantiation may not assign values to all type parameters, and those unassigned parameters stay as is. For instance, `f<T1, T2>` might have a partial instantiation `f<T1, u8>`.
- Once all partial instantiations are computed, the set is extended by unifying the instantiations against each other. If `<T>` and `<T'>` are in the set, and they unify under the substitution `s`, then `<s(T)>` will also be part of the set. For example, consider `f<T1, T2>` which modifies `M<T1>` and `R<T2>`, as well as accesses `M<u64>

and R<u8>. From this the instantiations <u64, T2> and <T1, u8> are computed, and the additional instantiation <u64, u8> will be added to the set.
- If after computing and extending instantiations any type parameters remain, they are skolemized into a given type as described earlier.

To understand the correctness of this procedure, consider the following arguments (a full formal proof is outstanding):

- *Direct interaction* Whenever a modified memory M<t> can influence the interpretation of M<t'>, a unifier must exist for the types t and t', and an instantiation will be verified which covers the overlap of t and t'.
- *Indirect interaction* If there is an overlap between two types which influences whether another overlap is semantically relevant, the combination of both overlaps will be verified via the extension step.

Notice that even though it is not common in regular Move code to work with both memory S<T> and, say, S<u64> in one function, there is a scenario where such code is implicitly created by injection of global invariants. Consider the example in Fig. 6. The invariant I1 which works on S<u64> is injected into the function g<R> which works on S<R>. When monomorphizing g, we need to verify an instance g_u64 in order to ensure that I1 holds.

4 Analysis

Reliability and Performance The three improvements described above resulted in a major qualitative change in performance and reliability. In the version of MVP released in September 2020, correct examples verified fairly quickly and reliably. But that is because we needed speed and reliability, so we disabled some properties that always timed out and others that timed out unpredictably when there were small changes in the framework. We learned that incorrect programs or specifications would time out predictably enough that it was a good bet that examples that timed out were erroneous. However, localizing the error to fix it was *very* hard, because debugging is based on a counterexample that violates the property, and getting a counterexample requires termination!

With each of the transformations described, we witnessed significant speedups and, more importantly, reductions in timeouts. Monomorphization was the last feature implemented, and, with it, timeouts almost disappeared. Although this was the most important improvement in practice, it is difficult to quantify because there have been many changes in Diem framework, its specifications, MVP, and even the Move language over that time.

It is simpler (but less important) to quantify the changes in run time of MVP on one of our more challenging modules, the DiemAccount module, which is the biggest module in the Diem framework. This module implements basic functionality to create and maintain multiple types of accounts on the blockchain, as well as manage their coin balances. It was called LibraAccount in release 1.0 of MVP, and is called DiemAccount today. The comparison requires various patches as described in [17]. The table below lists the consolidated numbers of lines, functions, invariants, conditions (requires, ensures, and aborts-if), as well as the verification times:

Module	Lines	Functions	Invariants	Conditions	Timing
LibraAccount	1975	72	10	113	**9.899s**
DiemAccount	2554	64	32	171	**7.340s**

Notice that `DiemAccount` has significantly grown in size compared to the older version. Specifically, additional specifications have been added. Moreover, in the original `LibraAccount`, some of the most complex functions had to be disabled for verification because the old version of MVP would time out on them. In contrast, in `DiemAccount` and with the new version, all functions are verified. Verification time has been improved by roughly 20%, *in the presence of three times more global invariants, and 50% more function conditions*.

We were able to observe similar improvements for the remaining of the 40 modules of the Diem framework. All of the roughly half-dozen timeouts resolved after introduction of the transformations described in this paper.

Causes for the Improvements It's difficult to pin down and measure exactly why the three transformations described improved performance and reliability so dramatically. We have explained some reasons in the subsections above: the alias-free memory model reduced search through combinatorial sharing arrangments, and the fine-grained invariant checking results in simpler formulas for the SMT solver.

We found that most timeouts in specifications stemmed from our liberal use of quantifiers. To disprove a property P_0 after assuming a list of properties, $P_1, \ldots p_n$, the SMT solver must show that $\neg P_0 \wedge P_1 \wedge \ldots \wedge P_n$ is satisfiable. The search usually involves instantiating universal quantifiers in $P_1, \ldots, P_n$. The SMT solver can do this endlessly, resulting in a timeout. Indeed, we often found that proving a post-condition false would time out, because the SMT solver was instantiating quantifiers to find a satisfying assignment of $P_1 \wedge \ldots \wedge P_n$. Simpler formulas result in fewer intermediate terms during solving, resulting in fewer opportunities to instantiate quantified formulas.

We believe that one of the biggest impacts, specifically on removing timeouts and improving predictability, is monomorphization. The reason for this is that monomorphization allows a multi-sorted representation of values in Boogie (and eventually the SMT solver). In contrast, before monomorphization, we used a universal domain for values in order to represent values in generic functions, roughly as follows:

```
type Value = Num(int) | Address(int) | Struct(Vector<Value>) | ...
```

This creates a large overhead for the SMT solver, as we need to exhaustively inject type assumptions (e.g. that a `Value` is actually an `Address`), and pack/unpack values. Consider a quantifier like `forall a: address: P(x)` in Move. Before monomorphization, we have to represent this in Boogie as `forall a: Value: is#Address(a)=> P(v#Address(a))`. This quantifier is triggered where ever `is#Address(a)` is present, independent of the structure of `P`. Over-triggering or inadequate triggering of quantifiers is one of the suspected sources of timeouts, as also discussed in [12].

Moreover, before monomorphization, global memory was indexed in Boogie by an address and a type instantiation. That is, for `struct R<T>` we would have one Boogie array `[Type, int]Value`. With monomorphization, the type index is eliminated, as we create different memory variables for each type instantiation. Quantification over memory content works now on a one-dimensional instead of an n-dimensional Boogie array.

Discussion and Related Work Many approaches have been applied to the verification of smart contracts; see e.g. the surveys [14,29]. [29] refers to at least two dozen systems for smart contract verification. It distinguishes between *contract* and *program* level approaches. Our approach has aspects of both: we address program level properties via pre/post conditions, and contract ("blockchain state") level properties via global invariants. To the best of our knowledge, among the existing approaches, the Move ecosystem is the first one where contract programming and specification language are fully integrated, and the language is designed from first principles influenced by verification. Methodologically, Move and the Move prover are thereby closer to systems like Dafny [11], or the older Spec# system [4], where instead of adding a specification approach posterior to an existing language, it is part from the beginning. This allows us not only to deliver a more consistent user experience, but also to make verification technically easier by curating the programming language.

In contrast to other approaches that only focus on specific vulnerability patterns [6,15,21,31], MVP offers a universal specification language. To the best of our knowledge, no existing specification approach for smart contracts based on inductive Hoare logic has similar expressiveness. We support universal quantification over arbitrary memory content, a suspension mechanism of invariants to allow non-atomic construction of memory content, and generic invariants. For comparison, the SMT Checker build into Solidity [8,9,10] does not support quantifiers, because it interprets programming language constructs (requires and assert statements) as specifications and has no dedicated specification language. While in Solidity one can simulate aspects of global invariants using modifiers by attaching pre/post conditions, this is not the same as our invariants, which are guaranteed to hold independent of whether a user may or (accidentally) may not attach a modifier, and which are optimized to be only evaluated as needed.

While the expressiveness of Move specifications comes with the price of undecidability and the dependency from heuristics in SMT solvers, MVP deals with this by its elaborated translation to SMT logic, as described in this paper. The result is a practical verification system that is fully integrated into the Diem blockchain production process, running in continuous integration, which is (to the best of our knowledge) a first in the industry.

The individual techniques we described are novel each by themselves. *Reference elimination* relies on borrow semantics, similar as in the Rust [16] language. We expect reference elimination to apply for the safe subset of Rust, though some extra work would be needed to deal with references aggregated by structs. However, we are not aware of that something similar has been attempted in existing Rust verification work [1,2,13,30]. *Global invariant injection* and the approach to minimize the number of assumptions and assertions is not applied in any existing verification approach we know of; however, we co-authored a while ago a similar line of work for *runtime checking* of invariants in Spec# [28], yet that work never left the conceptual state. *Monomorphization* is well known as a technique for compiling languages like C++ or Rust, where it is called specialization; however, we are not aware of it being generalized for modular verification of generic code where full type erasure is not possible, as it is the case in Move.

Future Work MVP is conceived as a tool for achieving higher assurance systems, not as a bug hunting tool. Having at least temporarily achieved satisfactory performance and reliability, we are turning our attention to the question of the goal of higher assurance, which raises several issues. If we're striving for high assurance, it would be great to be able to measure progress towards that goal. Since system requirements often stem from external business and regulatory needs, lightweight processes for exposing those requirements so we know what needs to be formally specified would be highly desirable.

As with many other systems, it is too hard to write high-quality specifications. Our current specifications are more verbose than they need to be, and we are working to require less detailed specifications, especially for individual functions. We could expand the usefulness of MVP for programmers if we could make it possible for them to derive value from simple reusable specifications. Finally, software tools for assessing the consistency and completeness of formal specifications would reduce the risk of missing bugs because of specification errors.

However, as more complex smart contracts are written and as more people write specifications, we expect that the inherent computational difficulty of solving logic problems will reappear, and there will be more opportunities for improving performance and reliability. In addition to translation techniques, it will be necessary to identify opportunities to improve SMT solvers for the particular kinds of problems we generate.

5 Conclusion

We described key aspects of the Move prover (MVP), a tool for formal verification of smart contracts written in the Move language. MVP has been successfully used to verify large parts of the Diem framework, and is used in continuous integration in production. The specification language is expressive, specifically by the powerful concept of global invariants. We described key implementation techniques which (as confirmed by our benchmarks) contributed to the scalability of MVP. One of the main areas of our future research is to improve specification productivity and reduce the effort of reading and writing specs, as well as to continue to improve speed and predictability of verification.

Acknowledgements This work would not have been possible without the many contributions of the Move platform team and collaborators. We specifically like to thank Bob Wilson, Clark Barrett, Dario Russi, Jack Moffitt, Jake Silverman, Mathieu Baudet, Runtian Zhou, Sam Blackshear, Tim Zakian, Todd Nowacki, Victor Gao, and Kevin Cheang.

References

1. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. PACMPL **3**(OOPSLA), 147:1–147:30 (2019)
2. Baranowski, M.S., He, S., Rakamaric, Z.: Verifying rust programs with SMACK. In: ATVA. Lecture Notes in Computer Science, vol. 11138, pp. 528–535. Springer (2018)
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387. Springer (2005)
4. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# Programming System: Challenges and Directions, pp. 144–152. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-69149-5_16
5. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Lecture Notes in Computer Science, vol. 7850, pp. 15–58. Springer (2013). https://doi.org/10.1007/978-3-642-36946-9_3
6. ConsenSys: Mythril Classic: Security analysis tool for Ethereum smart contracts, https://github.com/skylightcyber/mythril-classic
7. Dill, D.L., Grieskamp, W., Park, J., Qadeer, S., Xu, M., Zhong, J.E.: Fast and reliable formal verification of smart contracts with the move prover (extended version). CoRR **abs/2110.08362** (2021), https://arxiv.org/abs/2110.08362
8. Foundation, E.: Solidity documentation (2018), http://solidity.readthedocs.io
9. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. CoRR **abs/1907.04262** (2019)
10. Hajdu, Á., Jovanovic, D.: SMT-Friendly Formalization of the Solidity Memory Model. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 224–250. Springer (2020)
11. Leino, K.M.: Accessible software verification with dafny. IEEE Software **34**(06), 94–97 (nov 2017). https://doi.org/10.1109/MS.2017.4121212
12. Leino, K.R.M., Pit-Claudel, C.: Trigger Selection Strategies to Stabilize Program Verifiers. In: Proceedings of the 28th International Conference on Computer Aided Verification, Part I. pp. 361–381. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_20
13. Lindner, M., Aparicius, J., Lindgren, P.: No panic! verification of rust programs by symbolic execution. In: INDIN. pp. 108–114. IEEE (2018)
14. Liu, J., Liu, Z.: A survey on security verification of blockchain smart contracts. IEEE Access **7**, 77894–77904 (2019)
15. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
16. Matsakis, N.D., Klock, II, F.S.: The Rust Language, nourl = http://doi.acm.org/10.1145/2692956.2663188. Ada Lett. **34**(3), 103–104 (Oct 2014). https://doi.org/10.1145/2692956.2663188
17. Meng Xu: Artifact for Paper “Fast and Reliable Formal Verification of Smart Contracts with the Move Prover” (2020), https://github.com/meng-xu-cs/mvp-artifact
18. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (Oct 1992). https://doi.org/10.1109/2.161279
19. Morisander: The Biggest Smart Contract Hacks in History Or How to Endanger up to US $2.2 Billion. https://medium.com/solidified/the-biggest-smart-contract-hacks-in-history-or-how-to-endanger-up-to-us-2-2-billion-d5a72961d15d (2018)

20. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
21. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC. pp. 653–663. ACM (2018)
22. Sigalos, M.: Bug Puts $162 Million up for Grabs, Says Founder of DeFi Platform Compound. https://www.cnbc.com/2021/10/03/162-million-up-for-grabs-after-bug-in-defi-protocol-compound-.html (2021)
23. The CVC Team: CVC5, https://github.com/cvc5/cvc5
24. The Diem Association: An Introduction to Diem (2019), https://www.diem.com/en-us/
25. The Diem Association: The Diem Framework (2020), https://github.com/diem/diem/tree/release-1.5/diem-move/diem-framework
26. The Move Team: The Move Programming Language (2020), https://diem.github.io/move
27. The Move Team: The Move Specification Language (2020), https://github.com/diem/diem/blob/release-1.5/language/move-prover/doc/user/spec-lang.md
28. Tillmann, N., Grieskamp, W., Schulte, W.: Efficient checking of state-dependent constraints (US Patent 20050198621A1, 2004)
29. Tolmach, P., Li, Y., Lin, S., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. CoRR **abs/2008.02712** (2020), https://arxiv.org/abs/2008.02712
30. Toman, J., Pernsteiner, S., Torlak, E.: Crust: A bounded verifier for rust (N). In: ASE. pp. 75–80. IEEE Computer Society (2015)
31. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: ACM Conference on Computer and Communications Security. pp. 67–82. ACM (2018)
32. Zhong, J.E., Cheang, K., Qadeer, S., Grieskamp, W., Blackshear, S., Park, J., Zohar, Y., Barrett, C., Dill, D.L.: The Move Prover. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 137–150. Springer International Publishing (2020)

A Max-SMT Superoptimizer for EVM handling Memory and Storage*

Elvira Albert[1,2], Pablo Gordillo[2](✉),
Alejandro Hernández-Cerezo[2], and Albert Rubio[1,2]

[1] Instituto de Tecnología del Conocimiento, Madrid, Spain
[2] Complutense University of Madrid, Madrid, Spain
pabgordi@ucm.es

Abstract. Superoptimization is a compilation technique that searches for the optimal sequence of instructions semantically equivalent to a given (loop-free) initial sequence. With the advent of SMT solvers, it has been successfully applied to LLVM code (to reduce the number of instructions) and to Ethereum EVM bytecode (to reduce its gas consumption). Both applications, when proven practical, have left out memory operations and thus missed important optimization opportunities. A main challenge to superoptimization today is handling memory operations while remaining scalable. We present GASOL^{v2}, a gas and bytes-size superoptimization tool for Ethereum smart contracts, that leverages a previous Max-SMT approach for only stack optimization to optimize also *wrt.* memory and storage. GASOL^{v2} can be used to optimize the size in bytes, aligned with the optimization criterion used by the Solidity compiler solc, and it can also be used to optimize gas consumption. Our experiments on 12,378 blocks from 30 randomly selected real contracts achieve gains of 16.42% in gas *wrt.* the previous version of the optimizer without memory handling, and gains of 3.28% in bytes-size over code already optimized by solc.

1 Introduction and Related Work

Superoptimization is an automated technique for code optimization that was proposed back in 1987 [20]. It aims at automatically finding the *optimal* (*wrt.* the considered optimization criteria) instruction sequence —which is semantically equivalent— to a given sequence of loop-free instructions. It differs from traditional optimization techniques in that it uses search rather than applying pre-cooked transformations. However, as it requires exhaustive search in the space of valid instruction sequences, it suffers from high computation demands and it was considered impractical for many years. The first attempts of applying superoptimization were within a GNU C compiler back in the nineties [15] and, later, it has also been applied for an x86-64 assembly language [10, 11].

There is a recent revival of superoptimization due to the availability of SMT solvers which offer powerful techniques to handle enumerative search and

* This research was funded by the Spanish MCIN/AEI/10.13039/ 501100011033/-FEDER "Una manera de hacer Europa" projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union, and by the Ethereum Foundation grant FY21-0372.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 201–219, 2022.
https://doi.org/10.1007/978-3-030-99524-9_11

to check semantic equivalence. The approaches to supercompilation based on SMT can be roughly classified into two types: (1) Those that use an external synthesis algorithm with pruning techniques, such as [9,12,17], and that invoke the SMT solver to solve certain queries. This is the approach of the Souper superoptimizer [22] that relies on the synthesis algorithm for loop-free programs of Gulwani et al. [17]; (2) Those that directly produce an SMT encoding of the problem and use the search engine of the solver. This is the approach of [18], EBSO [21] and SYRUP [7]. Both types of approaches have been proven to be practical on their own settings and optimization criteria: the analysis of blocks does not reach the timeout of 10 sec in 90% of the cases [7] in SYRUP, and Souper optimized three million lines of C++ in 88 minutes [22]. The optimizations achieved vary for the considered criteria, Souper reported around 4.4% reduction in number of instructions, and SYRUP reported 0.58% in the global Ethereum gas usage. Scalability has been partly achieved because challenging features have been left out of the encoding: *memory operations have been excluded both in Souper and SYRUP*. While EBSO included a basic encoding for memory operations, its practicality was not proven: EBSO times out in 82% of the blocks and achieves optimization in less than 1% of all analyzed blocks. Leaving out memory operations dismisses optimization opportunities of two kinds: (a) as it works on smaller blocks of instructions (since the optimizer stops when finding a memory operation), the stack optimization is more limited, and (b) besides it misses possible optimizations on the memory operations themselves (e.g., eliminating unnecessary accesses).

The Ethereum Virtual Machine (EVM) has two areas where it can store items (besides the stack): (1) the *storage* is where all *contract state* variables reside, every contract has its own storage and it is persistent between external function calls (transactions) and has a higher gas cost to use; (2) the *memory* is used to hold temporary values, and it is erased between transactions and thus is cheaper to use. For conciseness, we often use "memory" to include both storage and memory, as their treatment for optimization is identical except for their associated costs. Our big challenge is to be able to handle memory operations while remaining practical, i.e., not reaching the timeout in the optimization of the vast majority of the blocks. This is achieved by leveraging SYRUP's two-staged method [7] to handle memory: (i) the first stage is devoted to synthesize a stack specification from the bytecode and apply simplification rules to it, and (ii) in a second stage a Max-SMT solver is used to perform the search for the optimal solution. When lifting such two-staged method to handle memory operations, we make two important extensions: in stage (i), we now synthesize a stack *and* memory specification from the bytecode on which we detect dependencies among memory operations and possibly remove redundant operations; (ii) this dependency information is included in our second stage as part of the encoding so that the SMT solver only needs to consider the dependence among such memory instructions when performing the search. Our two-staged approach allows isolating the dependency analysis process from the search itself, reducing the effort the SMT solver does in order to find the optimal sequence. The approach of Bansal and Aiken [10] to handle memory operations differs from ours on the superoptimization scope and the search process itself. Their tool considers multiple target sequences from a training set simultaneously and generates a database of (possibly) millions of optimizations. They enumerate all well-formed instructions sequences up to a certain size, including memory operations, and test the equivalence among them via a hash function. Our tool considers each sequence of instructions to optimize independently and the search is done via the search engine of an SMT solver.

GASOL^{v2} can be considered a successor of SYRUP [7], as it adopts its two-staged process and reuses part of its components, but it incorporates three fundamental extensions, and a new experimental evaluation, that constitute the main contributions of this paper: (1) GASOL^{v2} starts from the assembly json [1] generated by the solc compiler, rather than being used as a standalone optimization tool as SYRUP. This is fundamental to achieve a wide use of the tool since it is already linked to one of the most used compilers for coding Ethereum smart contracts. (2) It optimizes memory and storage operations using on one hand rule simplifications at the level of a specification synthesized from the bytecode, and on the other hand, a new SMT encoding which enables achieving a great balance between the accuracy and the overhead of the process. (3) While SYRUP is a tool that only optimizes the gas consumption of the bytecode, we have generalized some of its components to enable other optimization criteria. Currently we have included as well size in bytes, but other criteria can be easily incorporated now to the superoptimizer. (4) Besides we have performed a thorough experimental evaluation of our tool and have compared the results *wrt.* those obtained by SYRUP. The main conclusion of our evaluation is that handling memory operations in superoptimization pays off: it can achieve gains of 16.42% in gas over SYRUP, and reductions of 0.1% in gas and 3.28% in size (on already optimized code). If we assume that these savings are uniformly distributed, and the gas data obtained from Etherscan is constant, the 0.1% gas saved wrt the SYRUP [7] would amount nearly to 9.5 Million dollars in 2021.

GASOL^{v2} is part of the GASOL project [3], a GAS Optimization tooLkit for Ethereum smart contracts. The initial GASOL tool (i.e., GASOL^{v1}), presented in [5], aimed at detecting gas-expensive patterns within program loops (using resource analysis) and made a program transformation (which does not rely on SMT solvers) at the source code level. Hence, it contains a *global* (inter-block) optimization technique that is orthogonal to our superoptimizer, in which we perform *local* (or intra-block) transformations on loop-free code, and besides we work at bytecode rather than at source level. Both complementary techniques will be integrated within the GASOL toolkit, hence their names. In what follows, we drop $v2$ and use GASOL to refer to the tool presented in this paper.

2 The Architecture of GASOL

Figure 1 displays the architecture of GASOL, white components are borrowed from other tools, while gray components correspond to the new developments of this paper (either completely new, like **DEP**, or novel extensions for memory handling of previous SYRUP's implementations, like **SPEC**, **SIMP** and **SMS**).

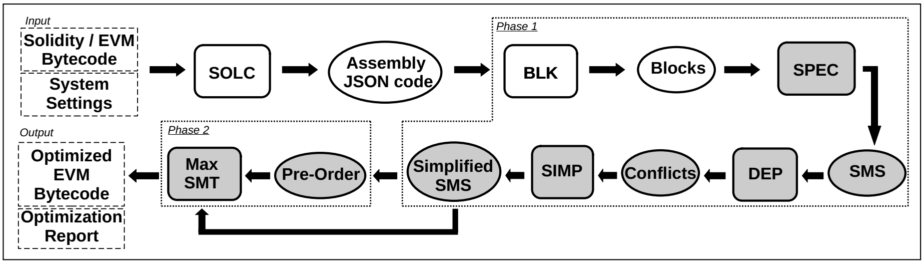


Fig. 1: Architecture of GASOL^{v2}

The **input** to GASOL is a smart contract (either its source in Solidity or its compiled EVM bytecode [23]), a selection of the optimization criteria (currently we are supporting gas consumption and size in bytes), and system settings (this includes compiler options for invoking the solc compiler and GASOL settings like the timeout per block of instructions). The **output** of GASOL is an optimized bytecode program and optionally a report with detailed information on the optimizations achieved (e.g., number of blocks optimized, number of blocks proven optimal, gas/size reduction gains, optimization time, among others).

The first component, labeled **SOLC** in the figure, invokes the Solidity compiler solc to obtain the bytecode in their assembly json exchange format [1]. Working on this exchange format has many advantages, one is that we can enable the optimizer of solc [4] and start the superoptimization from an already optimized bytecode. Besides, the format has been designed to be a usable common denominator for EVM 1.0, EVM 1.5 and Ewasm. Hence, we argue it is a good source for superoptimization as different target platforms will be able to use our tool equally. The assembly json format provides the EVM bytecode of the smart contract, metadata that relates it with the source Solidity code, and compilation information such as the version used to generate the bytecode. The output yield by GASOL can also be returned in assembly json format so that it can be used by other tools working on this format in the future. From the assembly json, the next component **BLK** partitions the bytecode given by solc into a set of sequences of loop-free bytecode instructions, named *blocks*, which correspond to the blocks of the CFG and also computes the size of the stack when entering each block.[3] We omit details of this step as it is standard in compiler construction and, for the case of the EVM, has been already subject of other analysis and optimization papers (see, e.g. [8,14,16] and their references).

The next component **SPEC** synthesizes a functional specification of the operand stack and of the memory and storage (SMS for short) for each block of bytecode instructions. This is done by symbolically executing the bytecodes in the block to extract from them what the contents of the operand stack and of the memory/storage are after executing them. The description of this component is given in Sec. 3.1. Next, **DEP** establishes the dependencies among the memory accesses from which a pre-order, that determines when a memory access needs

[3] In EVM, it is possible to reach a block with different stack sizes, and all such sizes can be statically computed. We will refer to the minimum or maximum when needed.

(1) $\tau(\texttt{MLOAD}, <\mathcal{S}, \mathcal{M}, \mathcal{S}t>) := <[\texttt{MLOAD}(\mathcal{S}[0])] + \mathcal{S}[1:n], \mathcal{M} + [\texttt{MLOAD}(\mathcal{S}[0])], \mathcal{S}t>$

(2) $\tau(\texttt{MSTORE}, <\mathcal{S}, \mathcal{M}, \mathcal{S}t>) := <\mathcal{S}[2:n], \mathcal{M} + [\texttt{MSTORE}(\mathcal{S}[0], \mathcal{S}[1])], \mathcal{S}t>$

(3) $\tau(\texttt{SLOAD}, <\mathcal{S}, \mathcal{M}, \mathcal{S}t>) := <[\texttt{SLOAD}(\mathcal{S}[0])] + \mathcal{S}[1:n], \mathcal{M}, \mathcal{S}t + [\texttt{SLOAD}(\mathcal{S}[0])]>$

(4) $\tau(\texttt{SSTORE}, <\mathcal{S}, \mathcal{M}, \mathcal{S}t>) := <\mathcal{S}[2:n], \mathcal{M}, \mathcal{S}t + [\texttt{SSTORE}(\mathcal{S}[0], \mathcal{S}[1])]>$

(5) $\tau(\texttt{SWAPX}, <\mathcal{S}, \mathcal{M}, \mathcal{S}t>) := \text{let temp} = \mathcal{S}[0] \;\; <\mathcal{S}[0/X][X/\text{temp}], \mathcal{M}, \mathcal{S}t>$

(6) $\tau(\texttt{POP}, <\mathcal{S}, \mathcal{M}, \mathcal{S}t>) := <\mathcal{S}[1:n], \mathcal{M}, \mathcal{S}t>$

Fig. 2: SMS Synthesis by Symbolic execution

to be performed before another one, is generated. For instance, subsequent load accesses, which are not interleaved by any store, do not have dependencies among them, while they do have with subsequent write accesses to the same positions. This phase is described in Secs. 3.2 (dependencies) and 3.3 (pre-order). In the next component **SIMP**, we apply simplification rules on the SMS. We include all stack simplification rules of SYRUP [7], as well as the additional rules we have developed for memory/storage simplifications. For instance, successive write accesses that overwrite the same memory position are simplified to a single one provided the same memory location is not read by any other instruction between them. The description of this component is given in Sec. 3.2. Finally, we generate a **Max-SMT** encoding from the (simplified) SMS that incorporates the pre-order established by the component **DEP** and from which the optimized bytecode is obtained. The description of this component is given in Sec. 4.

3 Synthesis of Stack and Memory Specifications

This section describes the first stage of the optimization (components **SPEC**, **SIMP** and **DEP**) that consists in synthesizing from a loop-free sequence of bytecode instructions a *simplified* specification of the stack and of the memory/storage (with the dependencies) that the execution of such bytecodes produces.

3.1 Initial Stack and Memory/Storage Specification

For each block, we synthesize its Stack and Memory Specification (SMS) by symbolically executing the instructions in the sequence. Function τ in Fig. 2 defines the symbolic execution for the memory/storage operations (1-4) and includes two representative stack opcodes (5-6). The first parameter of τ is a bytecode instruction and the second one is the SMS data structure $<\mathcal{S},\mathcal{M},\mathcal{S}t>$ whose first element corresponds to the stack ($\mathcal{S}$), the second one to the memory ($\mathcal{M}$), and the third one to the storage ($\mathcal{S}t$). The stack $\mathcal{S}$ is a list whose position $\mathcal{S}[0]$ corresponds to the top of the stack. At the beginning of executing a block, the stack contains the minimum number of elements needed to execute the block represented by symbolic variables s_i, where s_i models the element at $\mathcal{S}[i]$. The resulting list $\mathcal{M}$ ($\mathcal{S}t$ resp.) will contain the sequence of memory (storage resp.) accesses executed by the block. By abuse of notation, we often treat lists as sequences. Both $\mathcal{M}$ and $\mathcal{S}t$ are empty before executing the block symbolically. As an example, the symbolic execution of SSTORE removes the two top-most elements from $\mathcal{S}$, and adds the symbolic expression $\texttt{SSTORE}(\mathcal{S}[0], \mathcal{S}[1])$ to the storage sequence. Similarly, SLOAD removes from the top of the stack the position to be read, puts on the top of the stack the symbolic expression $\texttt{SLOAD}(\mathcal{S}[0])$ that represents the value read from the storage position $\mathcal{S}[0]$, and adds the same expression to the storage sequence $\mathcal{S}t$. As a result of applying τ to a sequence of bytecodes, the SMS obtained provides a specification of the target stack after executing the sequence in terms of the elements located in the stack before executing the sequence and, the target memory/storage (given as a sequence of accesses) after executing the sequence in terms of the input stack elements too.

Example 1. Consider the following bytecode that belongs to a real contract (bytecodes 0 to 47 of Welfare [2]). Its assembly json yield by the **SOLC** component contains 4524 bytecodes and after being partitioned by **BLK** we have 437 blocks to optimize. We illustrate the superoptimization of this block that contains in total 48 bytecodes from which 5 are the (underlined) memory/storage accesses:

```
1  PUSH1 80      9  DUP2         17  DUP4         25  PUSH2 3E8    33  PUSH2 FFFF   41  MUL
2  PUSH1 40     10  SLOAD        18  PUSH2 FFFF   26  PUSH1 1      34  MUL          42  OR
3  MSTORE       11  DUP2         19  AND          27  PUSH1 16     35  NOT          43  SWAP1
4  PUSH1 64     12  PUSH2 FFFF   20  MUL          28  PUSH2 100    36  AND          44  SSTORE
5  PUSH1 1      13  MUL          21  OR           29  EXP          37  SWAP1        45  POP
6  PUSH1 14     14  NOT          22  SWAP1        30  DUP2         38  DUP4         46  CALLVALUE
7  PUSH2 100    15  AND          23  SSTORE       31  SLOAD        39  PUSH2 FFFF   47  DUP1
8  EXP          16  SWAP1        24  POP          32  DUP2         40  AND          48  ISZERO
```

As **BLK** returns that the stack is empty when entering the block, we apply τ to the initial state $< [\,], [\,], [\,] >$ and produce the following SMS at the next selected lines:

$$L1 : \tau(\texttt{PUSH1 80}, < [\,], [\,], [\,] >) =< [128], [\,], [\,] >$$

$$L2 : \tau(\texttt{PUSH1 40}, < [128], [\,], [\,] >) =< [64, 128], [\,], [\,] >$$

$$L3 : \tau(\texttt{MSTORE}, < [64, 128], [\,], [\,] >) =< [\,], [\texttt{MSTORE(64,128)}], [\,] >$$

Finally, we get that at L48 $\mathcal{S} = [\texttt{ISZERO(CALLVALUE)}, \texttt{CALLVALUE}]$, $\mathcal{M} = [\texttt{MSTORE(64,128)}]$, $\mathcal{S}t = [\texttt{SLOAD}_1\texttt{(1)}, \texttt{SSTORE(1,V1)}, \texttt{SLOAD}_2\texttt{(1)}, \texttt{SSTORE(1,V2)}]$ where $\texttt{V1} = \texttt{OR}(\texttt{MUL}(...))$, $\texttt{AND}(\texttt{NOT}(..)), \texttt{SLOAD}_1\texttt{(1))}$ (omitting subexpressions) and `V2` is another similar expression involving arithmetic, binary operations and $\texttt{SLOAD}_2\texttt{(1)}$. Note that we use subscripts to distinguish the `SLOAD` instructions by their position in $\mathcal{S}t$. The stack specification contains a term that represents the result of the opcode `CALLVALUE` (executed at line 46, L46 for short), and a term with the result of executing the opcode `ISZERO` on `CALLVALUE`, stored on top of the stack. The memory only contains one element that is obtained by symbolically executing the three first instructions. The `PUSH` instructions at L1 and L2 introduce the values `64` and `128` on the stack, and the `MSTORE` executed at L3 introduces in $\mathcal{M}$ the symbolic expression `MSTORE(40,80)`. Similarly, $\mathcal{S}t$ contains the sequence of symbolic expressions that represent the storage instructions executed in the block at L10, L23, L31 and L44 respectively. The expressions corresponding to `V1` and `V2` are also obtained by applying function τ to the corresponding state. These stack expressions can be simplified in the next step using the rules in [7].

We note that the EVM memory is byte addressable (e.g., with instruction `MSTORE8`) and two different memory accesses may overlap. For simplicity of the presentation, we only consider the general case of word-addressable accesses, but the technique extends easily to the byte addressable case. In what follows, we use `LOAD` to abstract from the specific memory (`MLOAD`) and storage (`SLOAD`) bytecodes (and the same for `STORE`), when they are treated in the same way.

3.2 Memory/Storage Simplifications

In order to define the simplifications, and to later indicate to the SMT solver which memory instructions need to follow an order, we compute the conflicts between the different load and store instructions within each sequence.

Definition 1. *Two memory accesses A and B* conflict*, denoted as* conf(A,B) *if:*
(i) A is a store and B is a load and the positions they access might be the same;
(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

Note that in (ii) two store instructions that might operate on the same position do not conflict if the values they store are equal, as we will reach the same memory state regardless of the order in which the stores are executed. Note that two load instructions are never in conflict as the memory state does not change if we execute them in one order or another.

Given the SMS obtained in Sec. 3.1, we achieve simplifications by applying the stack simplification rules of [7] and, besides, the following new memory simplification rules based on Def. 1 to the M and S components (that achieve optimizations of type (b) according to the classification mentioned in Sec. 1):

Definition 2 (memory simplifications). *Let $< \mathcal{S}, \mathcal{M}, \mathcal{S}t >$ be an SMS, we can apply the following simplifications to any subsequence $b_1, \ldots, b_n$ in $\mathcal{M}$ or $\mathcal{S}t$:*
i) if $b_1 =$STORE(p, v) and $b_n =$LOAD(p) and $\nexists b_i =$STORE with $i \in \{2, \ldots, n-1\}$ and conf(b_1,b_i), we simplify it to $b_1, \ldots, b_{n-1}$ and replace b_n by v in the resulting SMS.
ii) if $b_1 =$STORE(p, v) and $b_n =$STORE(p, w) and $\nexists b_i =$LOAD with $i \in \{2, \ldots, n-1\}$ conf(b_1,b_i), we simplify it to $b_2, \ldots, b_n$.
iii) if $b_1 =$LOAD(p) and $b_n =$STORE$(p,$LOAD$(p))$ and $\nexists b_i =$STORE with $i \in \{2, \ldots, n-1\}$ conf(b_1,b_i), we simplify it to $b_1, \ldots, b_{n-1}$.

The simplifications can be applied in any order within $\mathcal{M}$ and $\mathcal{S}t$ until the process converges and the resulting sequence cannot be further simplified.

Intuitively, in (i), a load instruction from a position after a store instruction to the same position is simplified in the stack to the stored value provided there is no other store operation in between that might have changed the content of this position. In (ii), two subsequent store instructions to the same position are simplified to a single store if there is no load access on the same position between them. In (iii), a store instruction that stores in a position the result of the load in the same position can be removed, provided there is no other store in between that might have changed the content of this position. Note that such simplification rules can be applied to general-purpose compilers.

Example 2. In the SMS of Ex. 1, we have that conf(`SLOAD`$_1$`(1)`,`SSTORE(1,V1)`), conf(`SLOAD`$_1$`(1)`,`SSTORE(1,V2)`), conf(`SLOAD`$_2$`(1)`,`SSTORE(1,V1)`), conf(`SLOAD`$_2$`(1)`,`SSTORE(1,V2)`) and conf(`SSTORE(1,V1`,`SSTORE(1,V2)`) as all accesses operate on the same location. With these conflicts, we can apply rule i) to `SLOAD`$_2$`(1)`, as the previous `SSTORE` instruction has stored the value `V1` at the same location and there are no other storage instructions with conflict between them. Hence, we eliminate it from $\mathcal{S}t$ and replace it by `V1` in the resulting SMS. After that, we are able to apply rule ii) on the two `SSTORE` instructions as they store a value at the same position without conflict loads in between. Then, we remove `SSTORE(1,V1)` from $\mathcal{S}t$. The resulting SMS has the same $\mathcal{S}$ and $\mathcal{M}$ and $\mathcal{S}t$ is now [`SLOAD`$_1$`(1)`, `SSTORE(1,V2')`] where `V2'` is `V2` replacing `SLOAD`$_2$`(1)` by `V1`.

3.3 Pre-Order for Memory and Uninterpreted Functions

Given the SMS and using the conflict definition above, we generate a pre-order, as defined below, that indicates to the SMT solver the order between the memory accesses that needs to be kept in order to obtain the same memory state as the original one. Clearly, having more accurate conflict tests will result in weaker pre-orders and hence a wider search space for the SMT solver. This in turn will result in potentially larger optimization. Our implementation is highly parametric on the conflict test **DEP** so that more accurate tests can be easily incorporated.

Definition 3. *Let A and B be two memory accesses in a sequence S. We say that B has to be executed after A in S, denoted as $A \sqsubset B$ if:*

i) (store-store) B is a store instruction and A is the closest store instruction predecessor of B in S such that `conf(A,B)`*.*
ii) (load-store) A is a load instruction and B is the closest store instruction successor of A in S such that `conf(B,A)`*.*
iii) (store-load) B is a load instruction and A is the closest store instruction predecessor of B in S such that `conf(A,B)`*.*

Let us observe that we do not compute the closure for the dependencies at this stage, as the SMT solver will infer them, as explained in Sec. 4.2.

Example 3. From the simplified SMS of Ex. 2, we get the following load-store dependency, $\texttt{SLOAD}_1\texttt{(1)} \sqsubset \texttt{SSTORE(1,V2')}$, while the access `MSTORE(64,128)` has no dependencies as it is the unique memory operation.

Importantly, the notion of pre-order between memory instructions can also be naturally extended to all other operations that occur in the specification of the target stack. These operations are handled as uninterpreted functions and have to be called in the right order to build the result that is required in the target stack. Therefore, we propose a novel implementation (both in SYRUP and GASOL) that extends the pre-order $\sqsubset$ to uninterpreted functions by adding $A \sqsubset B$ also when:

iv) (uninterpreted-functions) A and B are uninterpreted functions that occur in the target stack as $B(\ldots, A(\ldots), \ldots)$.

While in the case of uninterpreted functions the pre-order is used for improving performance, for memory operations the use of the pre-order is mandatory for soundness, since it is what ensures that the obtained block after optimization has the same final state (in the stack, memory and storage) than the original block.

3.4 Bounding the Operations Position

As we will show in the next section, a solution to our SMT encoding assigns a position in the final instruction list to each operation such that the target stack is obtained. A key element for the performance of the encoding we propose in this paper is based on extracting from the instruction pre-order $\sqsubset$, upper and lower bounds to the position the operations can take in the instruction list. The lower bound for a given function is obtained by inspecting the subterm where it occurs in the target stack and analyzing its operands to detect the earliest point in which the result of all them can be placed in the stack, taking into account that shared subcomputations can be obtained using a `DUP` opcode. On the other hand, the upper bound for a function is obtained by inspecting the position in the target stack they occur and analyzing the operations that use the term that is headed by this function, to obtain the latest point in which this term could be computed. From this analysis, we obtain both the upper $UB(\iota)$ and lower $LB(\iota)$ bounds for every uninterpreted (which includes the load) and store operation ι, which are extensively used in the encoding provided in the next section.

4 Max-SMT Superoptimization

This section describes the second stage of the optimization process (named **Max SMT** in Fig. 1) that consists in producing, from the SMS and the dependencies, a Max-SMT encoding such that any valid model corresponds to a bytecode equivalent to the initial one and optimized for the selected criterion.

4.1 Stack Representation in the SMT Encoding

The stack representation is the same as in [7]: the stack can hold non-negative integer constants in the range $\{0, \ldots, 2^{256} - 1\}$, matching the 256-bit words in the EVM; initial stack variables $s_0, \ldots, s_{k-1}$, represent the initial (unknown) elements of the stack; and fresh variables $s_k, \ldots, s_v$ abstract each different subterm (built from opcodes and the initial stack variables) that appears in the SMS. A stack variable of the form s_i is represented in the encoding as the integer constant $2^{256} + i$, so that all stack elements in the model are integer values. To represent the contents of the stack after applying a sequence of instructions, a bound on the number of operations b_o and the size of the stack b_s must be given. These numbers are statically computed by considering the size of the initial block and the maximum number of stack elements involved. Then, propositional variables $u_{i,j}$, with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$, are used to denote whether there exists an element at position i in the stack after executing the first j operations, where the element $u_{0,j}$ refers to the topmost element of the stack. Quantified variables $x_{i,j} \in \mathbb{Z}$ are introduced to identify the word at position i after applying j operations, following the same format as $u_{i,j}$.

An instruction $\iota \in \mathcal{I}$ in the encoding can be either a basic stack opcode (`POP`, `SWAPk`, ...), a distinct expression that appears in the SMS or the extra instruction `NOP` that represents the possibility no opcode has been applied. A mapping θ is introduced to link every instruction in $\mathcal{I}$ to a non-negative integer in $\{0, \ldots, m_\iota\}$, where $m_\iota + 1 = |\mathcal{I}|$. This way, we can introduce the existentially quantified variables t_j, with $t_j \in \{0, \ldots, m_\iota\}$ and $j \in \{0, \ldots, b_o - 1\}$, to denote that the instruction ι is applied at step j when $t_j = \theta(\iota)$. There is a special case to be considered when identifying the instructions from an SMS: each expression containing a single occurrence of an opcode in W_{base} (see [23]) is considered as an independent expression with a different ι. Opcodes in W_{base} consume no operand from top of the stack and have lower gas cost and equal byte count as `DUPk`, so we can safely assume that in an optimal block such expressions are never duplicated. For efficiency reasons, we also apply the reciprocal: any other expression is forced to appear exactly once in our solution, as our experiments show that duplicating the expression is always better than computing it more than once. However, note that this may not hold, in general, when the cost of the expression is low or the size of the operating stack is high, and hence, although is highly unlikely, we may lose some better solutions. From this assumption, we have that every ι we have introduced must appear exactly once in every model, which simplifies greatly both the pre-order encoding and the gas model used. The following example illustrates how the SMS is processed and the relevance of considering W_{base}:

Example 4. Consider a modified version of Ex. 1, in which $\mathcal{S} = [$`ISZERO(CALLVALUE)`, `CALLVALUE`$]$ but $\mathcal{M}, \mathcal{S}t$ are both empty. b_0, b_s are bounded to 3 and 2 resp., as three instructions are enough to compute the given SMS and it reaches a stack size of two elements. Each application of `CALLVALUE` is considered independently, as `CALLVALUE` $\in W_{base}$. Variables $s_0 := 2^{256}, s_1 := 2^{256}+1, s_2 := 2^{256}+2$ are introduced to represent the stack variable obtained from $\texttt{CALLVALUE}_0, \texttt{CALLVALUE}_1$ and $\texttt{ISZERO(CALLVALUE}_1\texttt{)}$. GASOL creates the following θ map:

$$\theta := \{\texttt{PUSH} : 0, \texttt{POP} : 1, \texttt{NOP} : 2, \texttt{DUP1} : 3, \texttt{SWAP1} : 4,$$
$$\texttt{CALLVALUE}_0 : 5, \texttt{CALLVALUE}_1 : 6, \texttt{ISZERO(CALLVALUE}_1\texttt{)} : 7\}$$

The optimal sequence is `CALLVALUE CALLVALUE ISZERO`, which consumes 7 units of gas. It improves the cost of L46-L48, which consumes 8 due to the use of `DUP1`.

The set of instructions $\mathcal{I}$ can be split in four subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C \uplus \mathcal{I}_{St}$:

- $\mathcal{I}_S$ contains the basic stack operations: `PUSH`, `POP`, `NOP`, `DUP`k, and `SWAP`k, with $k \in \{1, \ldots, min(b_s - 1, 16)\}$. `DUP`$k$ and `SWAP`k are restricted by b_s because they cannot deal with elements that go beyond the maximum stack size.
- $\mathcal{I}_U$ contains the non-commutative uninterpreted functions that appear in the SMS. Its subset $\mathcal{I}_L \subseteq \mathcal{I}_U$ denotes the set of load instructions.
- $\mathcal{I}_C$ contains the commutative uninterpreted functions in the SMS.
- $\mathcal{I}_{St}$ contains the write operations in memory structures.

The encoding for subsets $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$ was already considered in [7], whereas $\mathcal{I}_{St}$ was left out. Instead, blocks were split when an opcode belonging to $\mathcal{I}_{St}$ was found. The inclusion of $\mathcal{I}_{St}$ instructions in the model leads to more savings in gas, as more optimizations can be applied in larger blocks (those correspond to optimizations of type (a) in the classification given in Sec. 1).

For each $\iota \in \mathcal{I}$ and each possible position j in the sequence of instructions, we add a constraint to represent the impact of this combination on the stack. These constraints match the semantics of τ when projecting onto the stack component, so that we encode the elements of the stack after executing ι in terms of the ones before its execution. They follow the structure $t_j = \theta(\iota) \Rightarrow C_\iota(j)$, where $C_\iota(j)$ expresses the changes in the stack after applying ι. The constraints for $\mathcal{I}_S \uplus \mathcal{I}_U \uplus \mathcal{I}_C$ are detailed in [7], our extension in this section is only to include the constraints to reflect the impact of storage operations on the stack. For this purpose, we use an auxiliary predicate *Move* (already used in [7]) to denote that all elements in the stack are moved two positions to the right in the resulting stack state. Thus, we have the following constraint for each position j and each

$\iota \in \mathcal{I}_{St}$, where o_0 and o_1 denote the position and value stored:

$$C_{St}(j,\iota) := \; t_j = \theta(\iota) \Rightarrow u_{0,j} \;\wedge\; u_{1,j} \;\wedge\; x_{0,j} = o_0 \;\wedge\; x_{1,j} = o_1 \;\wedge\; \mathit{Move}(j, 2, b_s - 1, -2) \;\wedge\; \neg u_{b_s-1,j+1} \;\wedge\; \neg u_{b_s-2,j+1}$$

Finally, we express the contents of the stack before executing the instructions of the block (initial stack) and after having executed them (target stack) by assigning the corresponding values (whether constants or stack variables) to $u_{i,0}, x_{i,0}$ and to u_{i,b_o}, x_{i,b_o} respectively. The overall SMT encoding for the stack representation is denoted as C_{SFS} and it is encoded using *QF_LIA* logic.

Example 5. Following Ex. 4, GASOL generates the constraint shown below to update the contents of the stack after applying $\iota =$ `ISZERO(CALLVALUE`$_1$`)` at step 2:

$$C_\iota(2,\iota) := \; t_2 = 7 \Rightarrow u_{0,2} \;\wedge\; x_{0,2} = 2^{256} + 1 \;\wedge\; u_{0,3} \;\wedge\; x_{0,3} = 2^{256} + 2 \;\wedge\; u_{1,3} = u_{1,2} \;\wedge\; x_{1,3} = x_{1,2}$$

4.2 Encoding the Pre-order Relation

Once the stack representation has been formalized, we also need to consider the conflicts that appear among memory operations as part of our encoding, as well as the dependencies between uninterpreted functions. All this is made by encoding the pre-order relation given in Sec. 3.3. We consider each pair of instructions ι, ι' s.t. $\iota \sqsubset \iota'$. We aim to prevent conflicting operations from appearing in the wrong order in a model, by imposing that ι cannot occur in the assignment after ι'.

Our proposed approach consists in introducing a variable $l_{\theta(\iota)}$ for every instruction $\iota \in \mathcal{I}_C \cup \mathcal{I}_U \cup \mathcal{I}_{St} := \mathcal{I}_{lord}$ to track the position it appears in a sequence. This information is useful for specifying multiple conditions in the encoding that are difficult to reflect otherwise. Firstly, these variables implicitly enforce that ι must be tied to exactly one position, and thus, included in every sequence exactly once. Besides, we can narrow the positions in which ι can appear by using $LB(\iota), UB(\iota)$ bounds. Finally, as *QF_LIA* supports ordering among variables, the order between conflicting instructions can be encoded as a plain comparison between their positions. Hence, the following constraints are derived:

$$L_P(\iota) := LB(\iota) \leq l_{\theta(\iota)} \leq UB(\iota) \;\wedge \bigwedge_{LB(\iota) \leq j \leq \, UB(\iota)} (l_{\theta(\iota)} = j) \Leftrightarrow (t_j = \theta(\iota))$$

$$L_{lord}(\iota, \iota') := l_{\theta(\iota)} < l_{\theta(\iota')} \;\; \text{where } \iota \sqsubset \iota'$$

Regarding memory operations, there is no need to consider special cases. The whole encoding can be expressed as follows:

$$C_{SMS} := \; C_{SFS} \;\wedge \bigwedge_{\iota \in \mathcal{I}_{lord}} L_P(\iota) \;\wedge \bigwedge_{\iota \sqsubset \iota'} L_{lord}(\iota, \iota')$$

4.3 Optimization using Max-SMT

As in [7], we formulate the problem of finding an optimal block as a partial weighted Max-SMT problem. In this section we show that the same encoding for gas optimization can be used in the presence of memory operations and that other optimization criterion, like bytes-size, can be included as well in our framework. Basically, in our Max-SMT problem, the *hard constraints* that must be satisfied by every model are those constraints for computing the SMS; and the *soft constraints* are used to find the optimal solution: a set of pairs $\{[C_1, \omega_1], \ldots, [C_n, \omega_n]\}$, where C_i denotes an SMT clause and ω_i its weight. The Max-SMT solver minimizes the weights of the falsified soft constraints. The weights of soft constraints presented in [7] match the gas spent for the sequence of instructions, thus ensuring an optimal model corresponds to a block that spends the least possible amount of gas. This gas encoding is also included in GASOL, but instructions in $\mathcal{I}_{lord}$ are removed from the *soft constraints*. Hard constraints already assert the exact number of times these instructions must appear in a sequence and therefore, they only add unnecessary extra cost that may harm the search of an optimality proof.

However, gas consumption is not the only relevant objective to consider when optimizing the code. When a contract is deployed, a fee of 200 units of gas must be paid for each non-zero byte of the EVM binary code. The desired trade-off between the initial deployment cost and invoking transactions can be specified in solc by setting the expected number of contract runs. In some cases, this leads to solc intentionally not fully replacing expressions that have a constant result by the value they represent if this constant is a large number, since the needed PUSH instructions will need many more non-zero bytes and hence will increment the deployment gas cost. For instance, if we want to have $2^{256} - 1$ on the top of the stack we can either push a zero and perform the bitwise NOT operation, which has gas cost 6 and non-zero bytes length 2 or push $2^{256} - 1$ directly which has gas cost 3 but non-zero bytes length 33.

When the bytes-size criterion is selected, we disable the application of the simplification rules of [7] that increase the byte-size and, besides, propose the next approach based on the *bytes-size model* for the Max-SMT encoding. This model is fairly simple except for the handling of the PUSH related instructions, denoted as $\mathcal{I}_P$ in what follows. All instructions that are not in $\mathcal{I}_P$ use exactly one byte. Instead PUSHx instructions take one byte to specify the opcode itself, and x bytes to include the pushed value. A first attempt to encode the weight of the PUSHx we tried was based on precisely describing the size in bytes based on the corresponding 32 options that x can take in terms of number of bytes. (recall that in EVM we have 256-bit words). This encoding is precise, but did not work in practice. An alternative, much simpler encoding, is based on the observation that numerical values can only appear in a model because at least once the corresponding PUSHx instruction is made. Later on, this value can be repeated using DUP, which has a minimal cost *wrt.* size of bytes, but if the block is large, some SWAP operation may also be needed. To make the encoding perform well in practice, we need to associate a single constant weight to all PUSHx operations, that is high enough to avoid models where expensive PUSHx operations are performed more than once instead of duplicating them. Our experiments have shown that a weight of 5 is enough to obtain optimal results for the sizes of blocks that the Max-SMT is able to handle. Then, we can assume NOP instructions cost 0 units, instructions in $\mathcal{I}_p$ costs 5 units and the remaining instructions cost 1 unit. Hence, three disjoint sets are introduced to match previous costs: $W_0 := \{\texttt{NOP}\}$,

$W_5 := \mathcal{I}_p$ and $W_1 := \mathcal{I}_S \setminus (W_0 \uplus W_5)$. Ω' bytes-size model is followed directly:

$$\Omega'_{SMS} := \bigcup_{0 \leqslant j < b_o} \{[t_j = \theta(\texttt{NOP}), 1], [\bigvee_{\iota \in W_0 \uplus W_1} t_j = \theta(\iota), 4]\}$$

Example 6. The optimized bytecode returned by GASOL for the gas criterion is `PUSH24* PUSH 80 PUSH 40 MSTORE PUSH 1 SLOAD PUSH32* AND PUSH21* OR PUSH32* AND OR PUSH 1 SSTORE CALLVALUE CALLVALUE ISZERO` (using * to skip large constants), which achieves a reduction of 5905 units *wrt.* the original version and is proven optimal. For the bytes-size criterion, GASOL times out due to the larger size of the block when size-increasing simplification rules are disabled. This issue will be discussed in Sec. 5.

5 Implementation and Experiments

This section provides further implementation details and describes our experimental evaluation. The GASOL tool is implemented in Python and uses as Max-SMT solver OptiMathSAT (OMS) [13] version 1.6.3 (which is the optimality framework of MathSAT). The aim of the experiments is to assess the effectiveness of our proposal by comparing it with the previous tool SYRUP. A timeout is given to the tools to specify the maximum amount of time that they can use for the analysis of each block. The timeout given to GASOL must be larger than for SYRUP because it works on less and larger blocks in order to analyze the same contract. We have used as timeout for SYRUP 10 sec, and for GASOL, we use 10*(#store+1) sec, as this would correspond to the addition of the times in SYRUP given to the partitioned blocks. It should be noted though that the cost of the search to be performed grows exponentially with the number of additional instructions. Therefore, in spite of giving a similar timeout, GASOL might time out in cases in which it has to deal with rather large blocks, while SYRUP does not on the corresponding smaller partitioned blocks. For this reason, we have implemented two additional versions: gasol$_{all}$ splits the blocks at all stores as SYRUP, and gasol$_{24}$ splits at store instructions only those blocks that have a size larger than 24 instructions. This is because we have observed during experimentation that the SMT search does not terminate in a reasonable time from that size on. The 24-partitioning starts from the end of the block and splits it if it finds a store. If the partitioned sub-block (from the start) still has a size larger than 24, further partitioning is done again if a new store is found from its end, and so on. Still, depending on where the stores are, the resulting blocks can have sizes larger than 24, as it happens in SYRUP as well. Further experimentation will be needed to come up with intelligent heuristics for the partitioning. The gasol versions implement all techniques described in the paper, including the SMT encoding dependencies between uninterpreted functions as described in Sec. 3.3. We have the following versions of GASOL and SYRUP in the evaluation: (1) syrup$_{cav}$ is the original tool from [7], (2) gasol$_{all}$ splits the blocks at all stores as in syrup$_{cav}$, (3) gasol$_{24}$ performs the 24-partitioning described above, (4) gasol$_{none}$ does not perform any additional partitioning of blocks, and (5) gasol$_{best}$ uses gasol$_{all}$, gasol$_{24}$, and gasol$_{none}$, as a portfolio of possible optimization results (running them in parallel) and keeps the best result.

We run the tools using the gas usage and the bytes-size criteria in Sec. 4.3. As already mentioned, SYRUP in [7] did not include the bytes-size criterion,

	$\mathbf{G}_{normal}$	$\mathbf{G}_{timeout}$	$\%\mathbf{G}_{total}$	$\mathbf{T}_{gas}$	$\mathbf{B}_{normal}$	$\mathbf{B}_{timeout}$	$\%\mathbf{B}_{total}$	$\mathbf{T}_{bytes}$
syrup$_{cav}$	35689	11129	0.62%	142,93	–	–	–	–
gasol$_{all}$	36344	11975	0.64%	120,21	3712	2213	2.64%	200,17
gasol$_{24}$	38765	12336	0.68%	327,36	4315	2238	2.92%	558,48
gasol$_{none}$	39977	0	0.53%	850,75	3871	0	1.72%	1194,38
gasol$_{best}$	41307	13197	0.72%	933,66	4676	2692	3.28%	1313,36

Table 2: Overall gains in gas and bytes-size and overheads

marked as "–" in the figures. Experiments have been performed on an Intel Core i7-7700T at 4.2GHz x 8 and 64Gb of memory, running Ubuntu 16.04.

The benchmark set. We have downloaded the last 30 verified smart contracts from Etherscan that were compiled using the version 8 of solc and whose source code was available as of June 21, 2021. The reason for this selection is twofold: (1) we require version 8 in order to be able to apply the latest solc optimizer and start from a worst-case scenario in which we have the most possible optimized version and, this way, assess if there is room for further optimization and, in particular, for the two types of gains achievable by GASOL (see Sec. 1), (2) we want to make a random choice (e.g., the last 30) rather than picking up contracts favorable to us. The benchmarks in [7] require using an old version of the compiler (at most 4), hence the last solc optimizer cannot be activated. The source code of GASOL as well as the smart contracts analyzed are available at https://github.com/costa-group/gasol-optimizer. We provide the results of analyzing the compiled smart contracts generated by the version 0.8.9 of solc with the complete optimization options. The total number of blocks, given by **BLK**, for the 30 contracts is 12,378. Within them, there are 1,044 `SSTORE` instructions, 6,631 `MSTORE` and 43 `MSTORE8`. These memory instructions are used by SYRUP to split the basic blocks, while GASOL does not split them always as explained above. This results on 15,416 blocks when considering the additional 24-partitioning, 13,030 without partitioning at stores by gasol$_{none}$, and 20,467 blocks by syrup$_{cav}$ and gasol$_{all}$. As in [7] all tools split blocks at instructions like `LOGX` or `CODECOPY` .

Efficiency gains and performance overhead. Table 2 shows the overall gas and size gains and the optimization time (in minutes). The total gas consumed by all contracts before running the optimizers is 7,538,907, and the bytes-size is 224,540. As it is customary, we are calculating such gas (resp. size) as the sum of the gas (resp. size) consumed by all EVM instructions in the considered contracts.[4] For those EVM instructions that do not consume a constant and fixed amount of gas, such as `SSTORE`, `EXP` or `SHA3`, we choose the lower bound that they may consume. Column $\mathbf{G}_{normal}$ refers to the gains for the blocks that do not timeout giving no solution, $\mathbf{G}_{timeout}$ represents the gas saved by the optimized blocks that reached the timeout in gasol$_{none}$ with no result (note that $\mathbf{G}_{normal}$ is the complementary of $\mathbf{G}_{timeout}$), and $\mathbf{G}_{total}$ the total gains computed as the sum of the previous two, given as a percentage *wrt.* the initial gas consumption. Columns **B** have the analogous meanings for size and **T** gives the time in minutes. The first observation is that our proposal of using dependencies in gasol$_{all}$ pays off, as we achieve larger

[4] Estimating the actual gains of executing transactions on the involved contracts is a research problem on its own which has been subject of other work, e.g., [6,16,19,24].

	#B	**Alr**$_g$	**Opt**$_g$	**Bet**$_g$	**Non**$_g$	**Tout**$_g$	**Alr**$_b$	**Opt**$_b$	**Bet**$_b$	**Non**$_b$	**Tout**$_b$
syrup$_{cav}$	20467	70.54	27.01	0.47	0.08	1.9	–	–	–	–	–
gasol$_{all}$	20467	70.63	27.36	0.64	0.35	1.02	83.25	12.83	1.2	0.69	2.03
gasol$_{24}$	15416	62.2	33.79	1.47	0.91	1.63	75.48	16.29	3.21	1.78	3.24
gasol$_{none}$	13030	65.48	25.3	3.81	0.34	5.07	73.44	11.7	3.1	2.57	9.19

Table 3: Optimization report (%) for SYRUP and GASOL

gains than syrup$_{cav}$ in less time. The second observation is that the gains in gas of GASOL are notably larger for blocks that do not time out $\mathbf{G}_{normal}$, as a larger search space can be explored. However, those blocks that would require a larger timeout might behave worse than the syrup$_{cav}$ and gasol$_{all}$ versions working on smaller blocks, as the original bytecode is taken as the optimization result in case of timeout. This sometimes happens in the version gasol$_{24}$, and more often in gasol$_{none}$. The problem is exacerbated for the bytes-size criterion because larger blocks are considered as a result of skipping size-increasing simplification rules. Even in $\mathbf{B}_{normal}$ the gain is smaller for gasol$_{none}$ than for gasol$_{24}$. This is because $\mathbf{B}_{normal}$ includes timeouts for which a solution is found. Our solution to mitigate the huge computation demands required in these cases is in row gasol$_{best}$ that runs in parallel gasol$_{all}$, gasol$_{24}$ and gasol$_{none}$ and returns the best result. As it can be seen, gasol$_{best}$ clearly outperforms the other systems in gas and size gains. As regards the overhead, it is also the most expensive option, as it reaches the timeout more often than the other systems and these timeouts are accumulated to the time. However, as superoptimizers are often used as offline optimization tools, which are run only prior to deployment, we argue that the gains compensate the further optimization time. Finally, it remains to be investigated the interaction between the two optimization criteria, namely how the reduction in bytes-size affects the gas consumption and vice versa.

Impact of phases 1 and 2. We would also like to estimate how much is gained in gasol$_{best}$ by applying the simplification rules and how much is gained by the SMT encoding. Regarding the simplification rules on memory, gasol$_{best}$ has applied 6 rules on storage and 11 on memory: 15 of them correspond to the rule i) (4 on storage and 11 on memory) described in Def. 2, and 2 to the rule ii) (both on storage). Rule iii) is never applied on this benchmark set, but we have applied it when optimizing other real smart contracts. As regards the percentage of the gains, 14.6% of the gas savings come from applying the memory rules, 34.4% from the stack rules and 51% is saved by the use of the Max-SMT solver. As in [7], the gains due to each phase are roughly half (i.e., 50% each). Regarding the simplification rules on stack for the gas criterion, their application has increased 11.4% in gasol$_{best}$ because it works on larger blocks and has more opportunities to apply them. However, when selecting the bytes-size criteria, there are less simplification rules applied (namely 96% less) as when the rules generate larger code in terms of size they are not applied (see Sec. 4.3).

Optimality results. Table 3 provides additional detailed information, which is also part of the *optimization report* of Fig. 1. Column **#B** shows the total number of blocks analyzed in each case, depending on the partitioning. In the remaining columns, we show the percentages of: Column **Alr** blocks that are already optimal, i.e., those blocks that cannot be optimized because they already consume the minimal amount of gas; **Opt** blocks that have been optimized and the SMT solver has proved the optimality of the solution, i.e., they consume the minimum amount of gas needed to generate the provided SMS; **Bet** blocks that have been optimized and therefore, consume less gas than the original ones, but the solution is not proved to be optimal; **Non** blocks that have not been optimized and the solver has not been able to prove if they are optimal, i.e., the solution found is the original one but it may exist a better one; **Tout** blocks where the solver reached the timeout without finding a model. The subscripts $_b$ are the analogous for the bytes-size criterion. We can observe in the table that gasol_{none} times out in more cases due to the larger sizes of the blocks that it optimizes, but the percentages of blocks for which it finds a better and optimal solution are notably high. It should also be noted that the results of SYRUP (and gasol_{all}) and, to a lesser extent, of gasol_{24} *wrt.* optimality are weaker. This is because they work on strictly smaller blocks and hence they can prove optimality for the partitioned blocks, but when glued together, the optimality may be lost. This is also the reason why the results for gasol_{best} are not included, because it mixes different notions of optimality and the concepts are not well-defined. Due to this weaker optimality, the **Opt** and **Bet** results are only slightly better for GASOL than for SYRUP. However, the truly important aspect is that the actual gas and size gains for GASOL in Table 2 are notably larger.

6 Conclusions and Future Work

We have presented GASOL^{v2}, a Max-SMT based superoptimizer for Ethereum smart contracts that uses the assembly json exchange format of the solc compiler for a direct integration into it. GASOL^{v2} extends the Max-SMT approach of SYRUP [7] with memory and storage operations, which constitute the most challenging and relevant features left out in SYRUP's approach. GASOL^{v2} is part of the GASOL project [3] that aims at developing a GAS Optimization tooLkit that will integrate *inter-block* optimizations [5] as well. Namely, the initial optimizer [5] of the GASOL project uses inter-block analysis to detect storage accesses that can be replaced by cheaper memory accesses, thus making global optimizations that are orthogonal and complementary to our intra-block ones. As part of our future work, we plan to investigate potential synergies among the different proposals to optimization for smart contracts. This includes also the cooperation with the solc optimizer [4] that incorporates classical compiler optimizations (e.g., dead code elimination, constant propagation, etc.) from which our superoptimizer is already benefiting (since we are applying the solc optimizer). In the other order of application, we expect also gains when applying classical analyses after superoptimization. For instance, we have also observed that after applying rule simplification (i) in Def. 2 and eliminating load instructions, we might leave store operations on memory locations that will never be accessed again, and that could be eliminated afterwards by applying an inter-block analysis ensuring that there are no further access to such memory location. The combination of the techniques and tools thus seems a promising direction for future research.

References

1. Compiler Input and Output JSON Description. https://docs.soliditylang.org/en/v0.8.7/using-the-compiler.html#compiler-input-and-output-json-description.
2. Welfare contract. https://etherscan.io/address/0x3E873439949793e8c577E08629c36Ed8c184e7D9#code.
3. GASOL – A GAS Optimization tooLkit, 2021. Funded by the Ethereum Foundation https://blog.ethereum.org/2021/07/01/esp-allocation-update-q1-2021/.
4. The solc optimizer, 2021. https://docs.soliditylang.org/en/v0.8.7/internals/optimizer.html.
5. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In Armin Biere and David Parker, editors, *Proceedings of 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020*, volume 12079 of *Lecture Notes in Computer Science*, pages 118–125, 2020.
6. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Don't run on fumes—parametric gas bounds for smart contracts. *Journal of Systems and Software*, 176:110923, 2021.
7. Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. Synthesis of super-optimized smart contracts using max-smt. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 177–200. Springer, 2020.
8. Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-Of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In *13th International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS 2019. Proceedings*, volume 11847 of *LNCS*, pages 63–78. Springer, 2019.
9. Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.
10. Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 394–403. ACM, 2006.
11. Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 177–192. USENIX Association, 2008.
12. James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 775–788. ACM, 2016.

13. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. Proceedings*, pages 93–107, 2013.
14. F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 127–137. IEEE, 2021.
15. Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 341–352. ACM, 1992.
16. Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.
17. Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011.
18. Abhinav Jangda and Greta Yorsh. Unbounded superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 78–88, 2017.
19. Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA*, volume 11247 of *LNCS*, pages 450–465. Springer, 2018.
20. Henry Massalin. Superoptimizer - A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987.
21. Julian Nagele and Maria A Schett. Blockchain superoptimizer. In *Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*, 2019. https://arxiv.org/abs/2005.05912.
22. Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]*, November 2017.
23. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2019.
24. Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*, pages 310–319, 2019.

Grammatical Inference

A New Approach for Active Automata Learning Based on Apartness*

Frits Vaandrager✉, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann

Institute for Computing and Information Sciences,
Radboud University, Nijmegen, the Netherlands
{f.vaandrager,b.garhewal,jrot,t.wissmann}@cs.ru.nl

Abstract. We present $L^{\#}$, a new and simple approach to active automata learning. Instead of focusing on equivalence of observations, like the L^* algorithm and its descendants, $L^{\#}$ takes a different perspective: it tries to establish *apartness*, a constructive form of inequality. $L^{\#}$ does not require auxiliary notions such as observation tables or discrimination trees, but operates directly on tree-shaped automata. $L^{\#}$ has the same asymptotic query and symbol complexities as the best existing learning algorithms, but we show that adaptive distinguishing sequences can be naturally integrated to boost the performance of $L^{\#}$ in practice. Experiments with a prototype implementation, written in Rust, suggest that $L^{\#}$ is competitive with existing algorithms.

Keywords: $L^{\#}$ algorithm · active automata learning · Mealy machine · apartness relation · adaptive distinguishing sequence · observation tree · conformance testing

1 Introduction

In 1987, Dana Angluin published a seminal paper [5], in which she showed that the class of regular languages can be learned efficiently using queries. In Angluin's approach of a *minimally adequate teacher (MAT)*, learning is viewed as a game in which a learner has to infer a deterministic finite automaton (DFA) for an unknown regular language L by asking queries to a teacher. The learner may pose two types of queries: "Is the word w in L?" (*membership queries*), and "Is the language recognized by DFA H equal to L?" (*equivalence queries*). In case of a *no* answer to an equivalence query, the teacher supplies a counterexample that distinguishes hypothesis H from L. The L^* algorithm proposed by Angluin [5] is able to learn L by asking a polynomial number of membership and equivalence queries (polynomial in the size of the corresponding canonical DFA).

Angluin's approach triggered a lot of subsequent research on active automata learning and has numerous applications in the area of software and hardware

[*] Research supported by NWO TOP project 612.001.852 "Grey-box learning of Interfaces for Refactoring Legacy Software (GIRLS)".

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 223–243, 2022.
https://doi.org/10.1007/978-3-030-99524-9_12

analysis, for instance for generating conformance test suites of software components [28], finding bugs in implementations of security-critical protocols [22,23,21], learning interfaces of classes in software libraries [33], inferring interface protocols of legacy software components [8], and checking that a legacy component and a refactored implementation have the same behavior [55]. We refer to [63,34] for surveys and further references.

Since 1987, major improvements of the original L^* algorithm have been proposed, for instance by [52,53,38,41,56,35,45,50,32,37,25]. Yet, all these improvements are variations of L^* in the sense that they approximate the Nerode congruence by means of refinement. Isberner [36] shows that these *descendants* of L^* can be described in a single, general framework.[1]

Variations of L^* have also been used as a basis for learning extensions of DFAs such as Mealy machines [48], I/O automata [2], non-deterministic automata [16], alternating automata [6], register automata [1,17], nominal automata [46], symbolic automata [40,7], weighted automata [14,11,30], Mealy machines with timers [64], visibly pushdown automata [36], and categorical generalisations of automata [62,29,12,18]. It is fair to say that L^*-like algorithms completely dominate the research area of active automata learning.

In this paper we present $L^{\#}$, a fresh approach to automata learning that differs from L^* and its descendants. Instead of focusing on equivalence of observations, $L^{\#}$ tries to establish *apartness*, a constructive form of inequality [61,26]. The notion of apartness is standard in constructive real analysis and goes back to Brouwer, with Heyting giving an axiomatic treatment in [31]. This change in perspective has several key consequences, developed and presented in this paper:

- $L^{\#}$ does not maintain auxiliary data structures such as observation tables or discrimination trees, but operates directly on the observation tree. This tree is a partial Mealy machine itself, and is very close to an actual hypothesis that can be submitted to the teacher. As a result, our algorithm is *simple*.
- The asymptotic query complexity of $L^{\#}$ is $\mathcal{O}(kn^2 + n \log m)$ and the asymptotic symbol complexity[2] is $\mathcal{O}(kmn^2 + nm \log m)$. Here k is the number of input symbols, n is the number of states, and m is the length of the longest counterexample. These are the *same asymptotic complexities* as the best existing (L^*-like) learning algorithms [52,53,32,37,36,25].
- The use of observation trees as primary data structure makes it easy to *integrate concepts from conformance testing to improve the performance* of $L^{\#}$. In particular, adaptive distinguishing sequences [39], which we can compute directly from the observation tree, turn out to be an effective boost in practice, even if their use does not affect asymptotic complexities. Through $L^{\#}$ testing and learning become even more intertwined [13,4].

[1] Except for the ZQ algorithm of [50], which was developed independently, and the ADT algorithm of [25], that was developed later and uses adaptive distinguishing sequences which are not covered in Isberner's framework.

[2] The symbol complexity is the number of input symbols required to learn an automaton. This is a relevant measure for practical learning scenarios, where the total time needed to learn a model is proportional to the number of input symbols.

- Experiments on benchmarks of [47], with a *prototype implementation* written in Rust, suggest that $L^{\#}$ is competitive with existing, highly optimized algorithms implemented in LearnLib [51].

Related work. Despite the different data structures, $L^{\#}$ and L^* [5] still have many similarities, since both store all the information gained from all queries so far. Moreover, both maintain a set of those states that have been learned with absolute certainty already. A few other algorithms have been proposed that follow a different approach than L^*. Meinke [43,44] developed a dual approach where, instead of starting with a maximally coarse approximating relation and refining it during learning, one starts with a maximally fine relation and coarsens it by merging equivalence classes. Although Meinke reports superior performance in the application to learning-based testing, these algorithms have exponential worst-case query complexities. Using ideas from [53], Groz et al. [27] use a combination of homing sequences and characterization sets to develop an algorithm for active model learning that does not require the ability to reset the system. Via an extensive experimental evaluation involving benchmarks from [47] they show that the performance of their algorithm is competitive with the L^* descendant of [56], but there can be huge differences in the performance of their algorithm for models that are similar in size and structure. Several authors have explored the use of SAT and SMT solvers for obtaining learning algorithms, see for instance [49,58], but these approaches suffer from fundamental scalability problems. In a recent paper, Soucha & Bogdanov [60] outline an active learning algorithm which also takes the observation tree as the primary data structure, and use results from conformance testing to speed up learning. They report that an implementation of their approach outperforms standard learning algorithms like L^*, but they have no explicit apartness relation and associated theoretical framework. It is precisely this theoretical underpinning which allowed us to establish complexity and correctness results, and define efficient procedures for counterexample processing and computing adaptive distinguishing sequences.

In the present paper, we first define partial Mealy machines, observation trees, and apartness (Section 2). Then, we present the full $L^{\#}$ algorithm (Section 3) and benchmark our prototype implementation (Section 4). The proofs of all theorems and complete benchmark results can be found in the appendix of the full version [65] of this paper.

2 Partial Mealy Machines and Apartness

The $L^{\#}$ algorithm learns a hidden (complete) Mealy machine, and its primary data structure is a *partial* Mealy machine. We first fix notation for partial maps.

We write $f\colon X \rightharpoonup Y$ to denote that f is a partial function from X to Y and write $f(x)\downarrow$ to mean that f is defined on x, that is, $\exists y \in Y\colon f(x) = y$, and conversely write $f(x)\uparrow$ if f is undefined for x. Often, we identify a partial function $f\colon X \rightharpoonup Y$ with the set $\{(x, y) \in X \times Y \mid f(x) = y\}$. The composition of partial maps $f\colon X \rightharpoonup Y$ and $g\colon Y \rightharpoonup Z$ is denoted by $g \circ f\colon X \rightharpoonup Z$, and we have

$(g \circ f)(x)\downarrow$ iff $f(x)\downarrow$ and $g(f(x))\downarrow$. There is a partial order on $X \rightharpoonup Y$ defined by $f \sqsubseteq g$ for $f, g\colon X \rightharpoonup Y$ if for all $x \in X$, $f(x)\downarrow$ implies $g(x)\downarrow$ and $f(x) = g(x)$.

Throughout this paper, we fix a finite set I of *inputs* and a set O of *outputs*.

Definition 2.1. *A* ***Mealy machine*** *is a tuple $\mathcal{M} = (Q, q_0, \delta, \lambda)$, where*

- *Q is a finite set of* ***states*** *and $q_0 \in Q$ is the* ***initial state****,*
- *$\langle \lambda, \delta \rangle\colon Q \times I \rightharpoonup O \times Q$ is a partial map whose components are an* ***output function*** *$\lambda\colon Q \times I \rightharpoonup O$ and a* ***transition function*** *$\delta\colon Q \times I \rightharpoonup Q$ (hence, $\delta(q,i)\downarrow \Leftrightarrow \lambda(q,i)\downarrow$, for $q \in Q$ and $i \in I$).*

We use superscript $\mathcal{M}$ to disambiguate to which Mealy machine we refer, e.g. $Q^{\mathcal{M}}$, $q_0^{\mathcal{M}}$, $\delta^{\mathcal{M}}$ and $\lambda^{\mathcal{M}}$. We write $q \xrightarrow{i/o} q'$, for $q, q' \in Q$, $i \in I$, $o \in O$ to denote $\lambda(q,i) = o$ and $\delta(q,i) = q'$. We call $\mathcal{M}$ ***complete*** *if δ is total, i.e., $\delta(q,i)$ is defined for all states q and inputs i. We generalize the transition and output functions to input words of length $n \in \mathbb{N}$ by composing $\langle \lambda, \delta \rangle$ n times with itself: we define maps $\langle \lambda_n, \delta_n \rangle\colon Q \times I^n \rightarrow O^n \times Q$ by $\langle \lambda_0, \delta_0 \rangle = \mathsf{id}_Q$ and*

$$\langle \lambda_{n+1}, \delta_{n+1} \rangle\colon \; Q \times I^{n+1} \xrightharpoonup{\langle \lambda_n, \delta_n \rangle \times \mathsf{id}_I} O^n \times Q \times I \xrightharpoonup{\mathsf{id}_{O^n} \times \langle \lambda, \delta \rangle} O^{n+1} \times Q$$

Whenever it is clear from the context, we use λ and δ also for words.

Definition 2.2. *The semantics of a state q is a map $[\![q]\!]\colon I^* \rightharpoonup O^*$ defined by $[\![q]\!](\sigma) = \lambda(q, \sigma)$. States q, q' in possibly different Mealy machines are* ***equivalent****, written $q \approx q'$, if $[\![q]\!] = [\![q']\!]$. Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are* ***equivalent*** *if their respective initial states are equivalent: $q_0^{\mathcal{M}} \approx q_0^{\mathcal{N}}$.*

In our learning setting, an *undefined* value in the partial transition map represents lack of knowledge. We consider maps between Mealy machines that preserve existing transitions, but possibly extend the knowledge of transitions:

Definition 2.3. *For Mealy machines $\mathcal{M}$ and $\mathcal{N}$, a* ***functional simulation*** *$f\colon \mathcal{M} \rightarrow \mathcal{N}$ is a map $f\colon Q^{\mathcal{M}} \rightarrow Q^{\mathcal{N}}$ with*

$$f(q_0^{\mathcal{M}}) = q_0^{\mathcal{N}} \qquad \text{and} \qquad q \xrightarrow{i/o} q' \text{ implies } f(q) \xrightarrow{i/o} f(q').$$

Intuitively, a functional simulation preserves transitions. In the literature, a functional simulation is also called *refinement mapping* [3].

Lemma 2.4. *For a functional simulation $f\colon \mathcal{M} \rightarrow \mathcal{N}$ and $q \in Q^{\mathcal{M}}$, we have $[\![q]\!] \sqsubseteq [\![f(q)]\!]$.*

For a given machine $\mathcal{M}$, an observation tree is simply a Mealy machine itself which represents the inputs and outputs we have observed so far during learning. Using functional simulations, we define it formally as follows.

Definition 2.5 ((Observation) Tree). *A Mealy machine $\mathcal{T}$ is a* ***tree*** *if for each $q \in Q^{\mathcal{T}}$ there is a unique sequence $\sigma \in I^*$ s.t. $\delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \sigma) = q$. We write $\mathsf{access}(q)$ for the sequence of inputs leading to q. A tree $\mathcal{T}$ is an* ***observation tree*** *for a Mealy machine $\mathcal{M}$ if there is a functional simulation $f\colon \mathcal{T} \rightarrow \mathcal{M}$.*

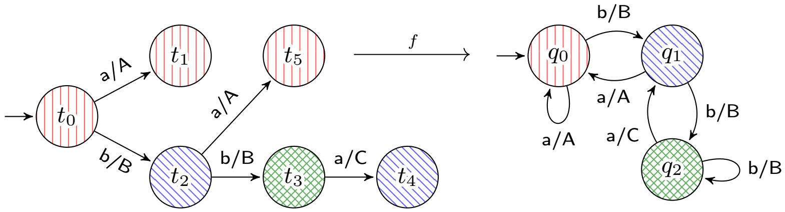


Fig. 1: An observation tree (left) for a Mealy machine (right).

Figure 1 shows an observation tree for the Mealy machine displayed on the right. The functional simulation f is indicated via coloring of the states.

By performing output and equivalence queries, the learner can build an observation tree for the unknown Mealy machine $\mathcal{M}$ of the teacher. However, the learner does not know the functional simulation. Nevertheless, by analysis of the observation tree, the learner may infer that certain states in the tree cannot have the same color, that is, they cannot be mapped to same states of $\mathcal{M}$ by a functional simulation. In this analysis, the concept of *apartness*, a constructive form of inequality, plays a crucial role [61,26]. A similar concept has previously been studied in the context of automata learning under the name *inequivalence constraints* in work on passive learning of DFAs, see for instance [15,24].

Definition 2.6. *For a Mealy machine $\mathcal{M}$, we say that states $q, p \in Q^{\mathcal{M}}$ are **apart** (written $q \# p$) if there is some $\sigma \in I^*$ such that $[\![q]\!](\sigma)\downarrow$, $[\![p]\!](\sigma)\downarrow$, and $[\![q]\!](\sigma) \neq [\![p]\!](\sigma)$. We say that σ is the **witness** of $q \# p$ and write $\sigma \vdash q \# p$.*

Note that the apartness relation $\# \subseteq Q \times Q$ is irreflexive and symmetric. A witness is also called *separating sequence* [59]. For the observation tree of Figure 1 we may derive the following apartness pairs and corresponding witnesses:

$$a \vdash t_0 \# t_3 \qquad a \vdash t_2 \# t_3 \qquad b\ a \vdash t_0 \# t_2$$

The apartness of states $q \# p$ expresses that there is a conflict in their semantics, and consequently, apart states can never be identified by a functional simulation:

Lemma 2.7. *For a functional simulation $f \colon \mathcal{T} \to \mathcal{M}$,*

$$q \# p \text{ in } \mathcal{T} \qquad \Longrightarrow \qquad f(q) \not\approx f(p) \text{ in } \mathcal{M} \qquad \text{for all } q, p \in Q^{\mathcal{T}}.$$

Thus, whenever states are apart in the observation tree $\mathcal{T}$, the learner knows that these are distinct states in the hidden Mealy machine $\mathcal{M}$.

The apartness relation satisfies a weaker version of *co-transitivity*, stating that if $\sigma \vdash r \# r'$ and q has the transitions for σ, then q must be apart from at least one of r and r', or maybe even both:

Lemma 2.8 (Weak co-transitivity). *In every Mealy machine $\mathcal{M}$,*

$$\sigma \vdash r \# r' \ \wedge\ \delta(q, \sigma)\downarrow \ \Longrightarrow\ r \# q \ \vee\ r' \# q \qquad \text{for all } r, r', q \in Q^{\mathcal{M}}, \sigma \in I^*.$$

We use the weak co-transitivity property during learning. For instance in Fig. 1, by posing the output query *aba*, consisting of the access sequence for t_1 concatenated with the witness *ba* for $t_0 \# t_2$, co-transitivity ensures that $t_0 \# t_1$ or $t_2 \# t_1$. By inspecting the outputs, the learner may conclude that $t_2 \# t_1$.

3 Learning Algorithm

The task solved by $L^{\#}$ is to find a strategy for the learner in the following game:

Definition 3.1. *In the learning game between a learner and a teacher, the teacher has a complete Mealy machine $\mathcal{M}$ and answers the following queries from the learner:*

OutputQuery(σ)**:** *For $\sigma \in I^*$, the teacher replies with the corresponding output sequence $\lambda^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma) \in O^*$.*[3]

EquivQuery$(\mathcal{H})$**:** *For a complete Mealy machine $\mathcal{H}$, the teacher replies* `yes` *if $\mathcal{H} \approx \mathcal{M}$ or* `no`*, providing some $\sigma \in I^*$ with $\lambda^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma) \neq \lambda^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma)$.*

Our $L^{\#}$ algorithm operates on an observation tree $\mathcal{T} = (Q, q_0, \delta, \lambda)$ for the unknown complete Mealy machine $\mathcal{M}$, where $\mathcal{T}$ contains the results of all output and equivalence queries so far. An observation tree is similar to the *cache* which is commonly used in implementations of L^*-based learning algorithms to store the answers to previously asked queries, avoiding duplicates [10,42]. But whereas for L^*-based learning algorithms the cache is an auxiliary data structure and only used for efficiency reasons, it is a first-class citizen in $L^{\#}$.

Remark 3.2. The learner has no information about the teacher's hidden Mealy machine. In particular, whenever we write #, we always refer to the apartness relation *on the observation tree* $\mathcal{T}$.

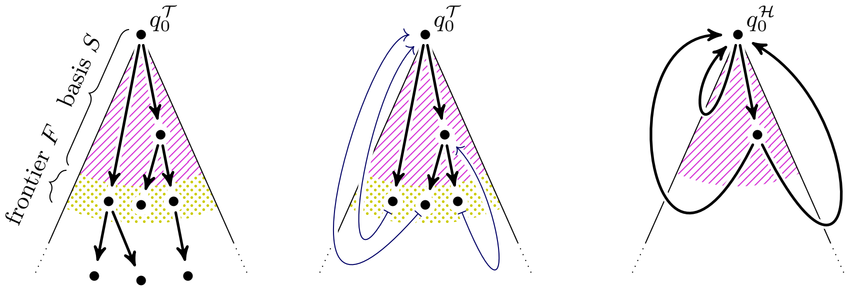
The observation tree is structured in a very similar way as Dijkstra's shortest path algorithm [19] structures a graph. Recall that during the execution of Dijkstra's algorithm 'the nodes are subdivided into three sets' [19]:

1. the nodes S to which a shortest path from the initial node is known. S initially only contains the initial node and grows from there.
2. the nodes F from which the next node to be added to S will be selected.
3. the remaining nodes.

This scheme adapts to the observation tree as follows and is visualized in Fig. 2a.

1. The states $S \subseteq Q^{\mathcal{T}}$, which already have been fully identified, i.e. the learner found out that these must represent distinct states in the teacher's hidden Mealy machine. We call S the *basis*. Initially, $S := \{q_0^{\mathcal{T}}\}$, and throughout the execution S forms a subtree of $\mathcal{T}$ and all states in S are pairwise apart: $\forall p, q \in S, p \neq q \colon p \# q$.

[3] In fact, later on we will assume that the teacher responds to slightly more general output queries to enable the use of *adaptive distinguishing sequences*, see Section 3.5.



(a) $\mathcal{T}$, S, and F (b) A choice $h\colon F \to S$ (c) Hypothesis $\mathcal{H}$ for h

Fig. 2: From the observation tree to the hypothesis ($|I| = 2$)

2. the *frontier* $F \subseteq Q^{\mathcal{T}}$, from which the next node to be added to S is chosen. Throughout the execution, F is the set of immediate non-basis successors of basis states: $F := \{q' \in Q \setminus S \mid \exists q \in S, i \in I : q' = \delta(q,i)\}$.
3. the remaining states $Q \setminus (S \cup F)$.

Initially, $\mathcal{T}$ consists of only an initial state $q_0^{\mathcal{T}}$ with no transitions. For every OutputQuery(σ) during the execution, the input $\sigma \in I^*$ and the corresponding response of type O^* is added automatically to the observation tree $\mathcal{T}$, and similarly every negative response to a EquivQuery leads to new states and transitions in the observation tree. With every extension $\mathcal{T}'$ of the observation tree $\mathcal{T}$, the apartness relation can only grow: whenever $p \# q$ in $\mathcal{T}$, then still $p \# q$ in $\mathcal{T}'$. Thus, along the learning game, $\mathcal{T}$ and $\#$ grow steadily:

Assumption 3.3 *We implicitly require that via output and equivalence queries, the observation tree $\mathcal{T}$ and the basis S are gradually extended, with the frontier F automatically moving along while S grows.*

3.1 Hypothesis construction

At almost any point during the learning game, the learner can come up with a hypothesis $\mathcal{H}$ based on the knowledge in the observation tree $\mathcal{T}$. Since the basis S contains the states already discovered, the set of states of such a hypothesis is simply set to $Q^{\mathcal{H}} := S$, and it contains every transition between basis states (in $\mathcal{T}$). The hypothesis must also reflect the transitions in $\mathcal{T}$ that leave the basis S, i.e. the transitions to the frontier. Those are resolved by finding for every frontier state a base state, for which the learner conjectures that they are equivalent states in the hidden Mealy machine. This choice boils down to a map $h\colon F \to S$ ($\mapsto$ in Fig. 2b). Then, a transition $q \xrightarrow{i/o} p$ in $\mathcal{T}$ with $q \in S$, $p \in F$ leads to a transition $q \xrightarrow{i/o} h(p)$ in $\mathcal{H}$ (Fig. 2c). These ideas are formally defined as follows.

Definition 3.4. *Let $\mathcal{T}$ be an observation tree with basis S and frontier F.*

1. *A Mealy machine $\mathcal{H}$* ***contains the basis*** *if $Q^{\mathcal{H}} = S$ and $\delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \mathsf{access}(q)) = q$ for all $q \in S$.*

2. *A* ***hypothesis*** *is a complete Mealy machine $\mathcal{H}$ containing the basis such that $q \xrightarrow{i/o'} p'$ in $\mathcal{H}$ ($q \in S$) and $q \xrightarrow{i/o} p$ in $\mathcal{T}$ imply $o = o'$ and $\neg(p \# p')$ (in $\mathcal{T}$).*
3. *A hypothesis $\mathcal{H}$ is* ***consistent*** *if there is a functional simulation $f\colon \mathcal{T} \to \mathcal{H}$.*
4. *For a Mealy machine $\mathcal{H}$ containing the basis, an input sequence $\sigma \in I^*$ is said to* ***lead to a conflict*** *if $\delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \sigma) \# \delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma)$ (in $\mathcal{T}$).*

Intuitively, the first three notions describe how confident we are in the correctness of the 'back loops' in $\mathcal{H}$ obtained from a choice $h\colon F \to S$. Notion 1 does not provide any warranty, notion 2 asserts that $\neg(q \# h(q))$ for all $q \in F$, and notion 3 (by definition) means that $\mathcal{T}$ is an observation tree for $\mathcal{H}$, that is, all observations so far are consistent with the hypothesis $\mathcal{H}$. The learner can verify the consistency of a hypothesis without querying the teacher (algorithm is in Section 3.3 below). The existence and uniqueness of a hypothesis are related to criteria on $\mathcal{T}$:

Definition 3.5. *In an observation tree $\mathcal{T}$, a state in F is 1.* ***isolated*** *if it is apart from all states in S and 2. is* ***identified*** *if it is apart from all states in S except one. 3. The basis S is* ***complete*** *if each state in S has a transition for each input in I.*

Lemma 3.6. *For an observation tree $\mathcal{T}$, if F has no isolated states then there exists a hypothesis $\mathcal{H}$ for $\mathcal{T}$. If S is complete and all states in F are identified then the hypothesis is unique.*

With a growing observation tree $\mathcal{T}$, the hidden Mealy machine is found as soon as the basis is big enough:

Theorem 3.7. *Suppose $\mathcal{T}$ is an observation tree for a (hidden) Mealy machine $\mathcal{M}$ such that S is complete, all states in F are identified, and $|S|$ is the number of equivalence classes of $\approx^{\mathcal{M}}$. Then $\mathcal{H} \approx \mathcal{M}$ for the unique hypothesis $\mathcal{H}$.*

The theorem itself is not necessary for the correctness of $L^{\#}$, but guarantees feasibility of learning.

3.2 Main loop of the algorithm

The $L^{\#}$ algorithm is listed in Algorithm 1 in pseudocode. The code uses Dijkstra's guarded command notation [20], which means that the following rules are applied non-deterministically until none of them can be applied anymore:

(R1) If F contains an isolated state, then this means that we have discovered a new state not yet present in S, hence we move it from F to S.

(R2) When a state $q \in S$ has no outgoing i-transition, for some $i \in I$, the output query for $\mathsf{access}(q)\ i$ will add the generated i successor, implicitly extending the frontier F.

Algorithm 1 Overall $L^{\#}$ algorithm

```
procedure LSharp
    do $q$ isolated, for some $q \in F$ $\rightarrow$                                  ▷ Rule (R1)
        $S \leftarrow S \cup \{q\}$
    [] $\delta^{\mathcal{T}}(q, i)\uparrow$, for some $q \in S, i \in I$ $\rightarrow$                         ▷ Rule (R2)
        OutputQuery(access($q$) $i$)
    [] $\neg(q \# r)$, $\neg(q \# r')$, for some $q \in F$, $r, r' \in S$, $r \neq r'$ $\rightarrow$   ▷ Rule (R3)
        $\sigma \leftarrow$ witness of $r \# r'$
        OutputQuery(access($q$) $\sigma$)
    [] $F$ has no isolated states and basis $S$ is complete $\rightarrow$            ▷ Rule (R4)
        $\mathcal{H} \leftarrow$ BuildHypothesis
        $(b, \sigma) \leftarrow$ CheckConsistency($\mathcal{H}$)
        if $b =$ yes then
            $(b, \rho) \leftarrow$ EquivQuery($\mathcal{H}$)
            if $b =$ yes then: return $\mathcal{H}$
            else: $\sigma \leftarrow$ shortest prefix of $\rho$ such that $\delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma) \# \delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \sigma)$ (in $\mathcal{T}$)
        end if
        ProcCounterEx($\mathcal{H}, \sigma$)
    end do
end procedure
```

(R3) When $q \in F$ is a state in the frontier that is not yet identified, then there are at least two states in S that are not apart from q. In this case, the algorithm picks a witness $\sigma \in I^*$ for $r \# r'$. After the OutputQuery(access(q) σ), the observation tree is extended and thus q will be apart from at least r or r' by weak co-transitivity (Lemma 2.8).

(R4) When F has no isolated states and S is complete, BuildHypothesis picks a hypothesis $\mathcal{H}$ (at least one exists Lemma 3.6). If $\mathcal{H}$ is not consistent with observation tree $\mathcal{T}$ we get a conflict σ for free. Otherwise, we pose an equivalence query for $\mathcal{H}$. If the hypothesis is correct, $L^{\#}$ terminates, and otherwise we obtain a counterexample ρ. The counterexample decomposes into two words $\sigma\eta$, where σ leads to a conflict and η witnesses it. The conflict σ means that one of the frontier states was merged with an apart basis state in $\mathcal{H}$, causing a wrong transition in $\mathcal{H}$. Since σ can be very long, the task of ProcCounterEx(σ) is to shorten σ until we know which frontier state caused the conflict. So after ProcCounterEx, $\mathcal{H}$ is not a hypothesis for the updated $\mathcal{T}$ anymore.

We will show the correctness of $L^{\#}$ in a top-down approach discussing the subroutines later and only assuming now that:

1. BuildHypothesis picks one of the possible hypotheses (Lemma 3.6)
2. CheckConsistency($\mathcal{H}$) tells if there is a functional simulation $\mathcal{T} \rightarrow \mathcal{H}$, and if not, provides $\sigma \in I^*$ leading to a conflict (Lemma 3.10 below).
3. If $\mathcal{H}$ contains the basis and σ leads to a conflict, then ProcCounterEx($\mathcal{H}, \sigma$), extends $\mathcal{T}$ such that $\mathcal{H}$ is not a hypothesis anymore (Lemma 3.11 below).

Whenever the algorithm terminates, the learner has found the correct model. Therefore, correctness amounts to showing termination. The rough idea is that each rule will let S, F, or $\#$ restricted to $S \times F$ grow, and each of these sets are bounded by the hidden Mealy machine $\mathcal{M}$. We define the norm $N(\mathcal{T})$ by

$$\frac{|S| \cdot (|S|+1)}{2} \; + \; |\{(q,i) \in S \times I \mid \delta^{\mathcal{T}}(q,i)\downarrow\}| \; + \; |\{(q,q') \in S \times F \mid q \# q'\}| \quad (1)$$

The first summand increases whenever a state is moved from F to S (R1); it is quadratic in $|S|$ because (R1) reduces the third summand. The second summand records the progress achieved by extending the frontier (R2). The third summand counts how much the states in the frontier are identified (R3). Rule (R4) extends the apartness relation, leading to an increase of the third summand.

Theorem 3.8. *Every rule application in $L^{\#}$ increases the norm $N(\mathcal{T})$ in (1).*

The norm $N(\mathcal{T})$ and therefore also the number of rule applications is bounded:

Theorem 3.9. *If $\mathcal{T}$ is an observation tree for $\mathcal{M}$ with n equivalence classes of states and $|I| = k$, then $N(\mathcal{T}) \leq \frac{1}{2} \cdot n \cdot (n+1) + kn + (n-1)(kn+1) \in \mathcal{O}(kn^2)$.*

At any point of execution, either rule (R1), (R2), or (R4) is applicable, so $L^{\#}$ never blocks. As soon as the norm $N(\mathcal{T})$ hits the bound, the only applicable rule is rule (R4) with the teacher accepting the hypothesis. Thus, the correct Mealy machine is learned within $\mathcal{O}(k \cdot n^2)$ rule applications. The complexity in terms of the input parameters is studied in Section 3.6.

We now continue defining the subroutines and proving them correct.

3.3 Consistency checking

A hypothesis $\mathcal{H}$ is not necessarily *consistent* with $\mathcal{T}$, in the sense of a functional simulation $\mathcal{T} \to \mathcal{H}$. Via a breadth-first search of the Cartesian product of $\mathcal{T}$ and $\mathcal{H}$ (Algorithm 2), we may check in time linear in the size of $\mathcal{T}$ whether a functional simulation $\mathcal{T} \to \mathcal{H}$ exists. In the negative case, we obtain $\sigma \in I^*$ leading to a conflict without any equivalence or output query to the teacher needed. Thus, this is also called 'counterexample milking' [10].

Lemma 3.10. *Algorithm 2 terminates and is correct, that is, if $\mathcal{H}$ is a hypothesis for $\mathcal{T}$ with a complete basis, then* CheckConsistency($\mathcal{H}$)

1. *returns* `yes`*, if $\mathcal{H}$ is consistent,*
2. *returns* `no` *and $\rho \in I^*$, if ρ leads to a conflict ($\delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \rho) \# \delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \rho)$ in $\mathcal{T}$).*

3.4 Counterexample processing

The L^* algorithm [5] performs $\mathcal{O}(m)$ queries to analyze a counterexample of length m. So if a teacher returns really long counterexamples, their analysis will dominate the learning process. Rivest & Schapire [52,53] improve counterexample

Algorithm 2 Check if hypothesis $\mathcal{H}$ is consistent with observation tree $\mathcal{T}$

```
procedure CheckConsistency($\mathcal{H}$)
    $Q \leftarrow$ new $queue \subseteq S \times S$
    $enqueue(Q, (q_0^{\mathcal{T}}, q_0^{\mathcal{H}}))$
    while $(q, r) \leftarrow dequeue(Q)$
        if $q \# r$ then: return no: access($q$)
        for all $q \xrightarrow{i/o} p$ in $\mathcal{T}$ do
            $enqueue(Q, (p, \delta^{\mathcal{H}}(r, i)))$
        end for
    end while
    return yes
end procedure
```

analysis of L^* using binary search, requiring only $\mathcal{O}(\log m)$ queries. A similar trick is applied in $L^{\#}$.

Suppose σ leads to a conflict $q \# r$ for $q = \delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma)$ and $r = \delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \sigma)$. Then, ProcCounterEx($\sigma$) (Algorithm 3) extends $\mathcal{T}$ such that $\mathcal{H}$ will never be a hypothesis for $\mathcal{T}$ again.

If $r \in S \cup F$, then the conflict $q \# r$ is obvious and $\mathcal{H}$ is not a hypothesis again. If otherwise $r \notin S \cup F$, the binary search will successively reduce the number of transitions of σ outside $S \cup F$ by a factor of 2 until we reach the above base case $S \cup F$. Let $\sigma_1 \sigma_2 := \sigma$ such that the run of σ_1 in $\mathcal{T}$ ends halfway between the frontier and r. By an additional output query, the binary search checks whether already σ_1 leads to a conflict. In the two cases, we can either avoid σ_1 or σ_2, so we reduce the number of transitions outside $S \cup F$ to half the amount. The precise argument is in:

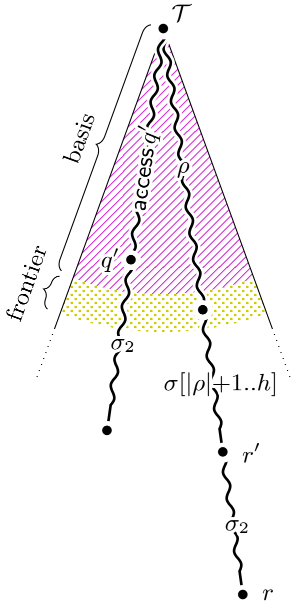
Lemma 3.11. *Suppose basis S is complete, $\mathcal{H}$ is a complete Mealy machine containing the basis, and $\sigma \in I^*$ leads to a conflict. Then* ProcCounterEx($\mathcal{H}, \sigma$) *terminates, performs at most $\mathcal{O}(\log_2 |\sigma|)$ output queries and is correct: upon termination, the machine $\mathcal{H}$ is not a hypothesis for $\mathcal{T}$ anymore.*

3.5 Adaptive distinguishing sequences

As an optimization in practice, we may extend the rules (R2) and (R3) by incorporating *adaptive distinguishing sequences* (Ads) into the respective output queries. Adaptive distinguishing sequences, which are commonly used in the area of conformance testing [39], are input sequences where the choice of an input may depend on the outputs received in response to previous inputs. Thus, strictly speaking, an ADS is a decision graph rather than a sequence. This mild extension of the learning framework reflects the actual black box behaviour of Mealy machines: for every input in I sent to the hidden Mealy machine, the learner observes the output O before sending the next input symbol. Use of adaptive distinguishing sequences may reduce the number of output queries that are required for the identification of frontier states.

Algorithm 3 Processing σ that leads to a conflict, i.e. $\delta^{\mathcal{H}}(q_0,\sigma) \# \delta^{\mathcal{T}}(q_0,\sigma)$

```
procedure ProcCounterEx($\mathcal{H}$, $\sigma \in I^*$)
    $q \leftarrow \delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma)$
    $r \leftarrow \delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \sigma)$
    if $r \in S \cup F$ then
        return
    else
        $\rho \leftarrow$ unique prefix of $\sigma$ with $\delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \rho) \in F$
        $h \leftarrow \lfloor \frac{|\rho|+|\sigma|}{2} \rfloor$
        $\sigma_1 \leftarrow \sigma[1..h]$
        $\sigma_2 \leftarrow \sigma[h+1..|\sigma|]$
        $q' \leftarrow \delta^{\mathcal{H}}(q_0^{\mathcal{H}}, \sigma_1)$
        $r' \leftarrow \delta^{\mathcal{T}}(q_0^{\mathcal{T}}, \sigma_1)$
        $\eta \leftarrow$ witness for $q \# r$
        OutputQuery($\mathsf{access}(q')\ \sigma_2\ \eta$)
        if $q' \# r'$ then
            ProcCounterEx ($\mathcal{H}$, $\sigma_1$)
        else
            ProcCounterEx ($\mathcal{H}$, $\mathsf{access}(q')\ \sigma_2$)
        end if
    end if
end procedure
```



As an example, consider the observation tree of Figure 3(left). The basis for this tree consists of 5 states, which are pairwise apart (separating sequences are a, ab and aa). Frontier states can be identified by the *single* adaptive sequence of Figure 3(right). The ADS starts with input a. If the response is 2 we have identified our frontier state as t_4. If the response is 0 then the frontier state is either t_0 or t_2, and we may identify the state with a subsequent input a. Similarly, if the response is 1 then the frontier state is either t_1 or t_3, and we may identify the state by a subsequent input b. We can therefore identify (or isolate) frontier state t_5 with a single (extended) output query that starts with the access sequence for t_5 ($bbbba$) followed by the ADS of Figure 3(right). If we used separating sequences, we would need at least 2 output queries.

In the setting of $L^{\#}$, we can directly compute an optimal ADS from the current observation tree. To this end, we recursively define an *expected reward* function E, which sends a set $U \subseteq Q^{\mathcal{T}}$ of states to the maximal expected number of apartness pairs (in the absence of unexpected outputs).

$$E(U) = \max_{i \in inp(U)} \left(\sum_{o \in O} \frac{|U \xrightarrow{i/o}| \cdot (|U \xrightarrow{i}| - |U \xrightarrow{i/o}| + E(U \xrightarrow{i/o}))}{|U \xrightarrow{i}|} \right) \tag{2}$$

where $inp(U) := \{i \in I \mid \exists q \in U : \delta^{\mathcal{T}}(q,i)\downarrow\}$, $U \xrightarrow{i} := \{q \in U \mid \delta^{\mathcal{T}}(q,i)\downarrow\}$ and $U \xrightarrow{i/o} := \{q' \in Q^{\mathcal{T}} \mid \exists q \in U : q \xrightarrow{i/o} q'\}$. We define the maximum over the empty set to be 0. Then Ads(U) is the decision tree constructed as follows:

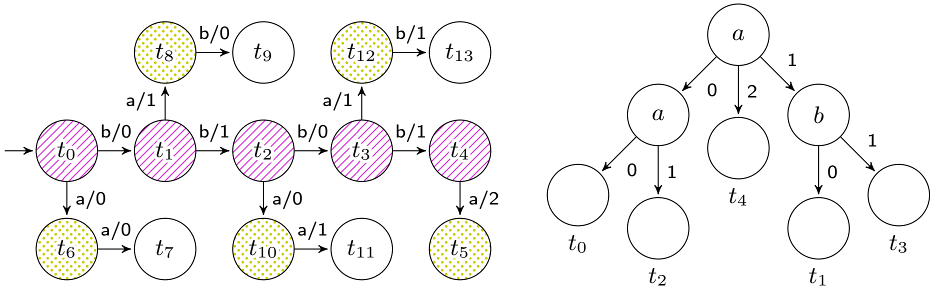


Fig. 3: An observation tree (left) and an ADS for its basis (right)

- If $U \xrightarrow{i} = \emptyset$ then $\text{Ads}(U)$ consists of a single node U without a label.
- If $U \xrightarrow{i} \neq \emptyset$ then $\text{Ads}(U)$ is constructed by choosing an input i that witnesses the maximum $E(U)$, creating a node U with label i, and, for each output o with $U \xrightarrow{i/o} \neq \emptyset$, adding an o-transition to $ADS(U \xrightarrow{i/o})$.

For the observation tree of Figure 3(left) we may compute $E(\{t_0, \dots, t_4\}) = 4$ and obtain the decision tree of Figure 3(right) as ADS. Running the ADS from state t_5 will create 4 new apartness pairs with basis states (or 5 in case an unexpected output occurs, e.g. $a(1)b(2)$).

Proposition 3.12. *Define $L^{\#}_{\text{Ads}}$ by replacing the output queries in $L^{\#}$ with*

(R2') $\text{OutputQuery}(\mathsf{access}(q)\ i\ \text{Ads}(S))$ *in (R2) and*
(R3') $\text{OutputQuery}(\mathsf{access}(q)\ \text{Ads}(\{b \in S \mid \neg(b \# q)\}))$ *in (R3).*

Then, $L^{\#}_{\text{Ads}}$ lets the norm $N(\mathcal{T})$ grow for each rule application and thus is correct.

3.6 Complexity

Since equivalence queries are costly in practice and since processing of long counterexamples of length m requires $\mathcal{O}(\log m)$ output queries, it makes sense to postpone equivalence queries as long as possible:

Definition 3.13. ***Strategic*** *$L^{\#}$ (resp. $L^{\#}_{\text{Ads}}$) is the special case of Algorithm 1 where rule (R4) is only applied if none of the other rules is applicable.*

Then we obtain the following query complexity for the $L^{\#}$ algorithm.

Theorem 3.14. *Strategic $L^{\#}$ (resp. $L^{\#}_{\text{Ads}}$) learns the correct Mealy machine within $\mathcal{O}(kn^2 + n\log m)$ output queries and at most $n-1$ equivalence queries.*

The query complexity of $L^{\#}$ equals the best known query complexity for active learning algorithms, as achieved by Rivest & Schapire's algorithm [52,53], the observation pack algorithm [32], the TTT algorithm [37,36], and the ADT algorithm [25].

In a black box learning setting in practice, answering an output query for $\sigma \in I^*$ grows linearly with the length σ. Therefore, the (asymptotic) total number of input symbols sent by the learner is also a metric for comparing learning algorithms:

Theorem 3.15. *Let $n \in \mathcal{O}(m)$. Then the strategic $L^\#$ algorithm learns the correct Mealy machine with $\mathcal{O}(kmn^2 + nm \log m)$ input symbols.*

This matches the asymptotic symbol complexity of the best known active learning algorithms. Although ProcCounterEx reduces the length of the sequence leading to the conflict, the witness of the conflict remains of size $\Theta(m)$ in the worst case. This means that we need $\mathcal{O}(m \log m)$ symbols to process a single counterexample and $\mathcal{O}(nm \log m)$ symbols to process all counterexamples.

4 Experimental Evaluation

In the previous sections, we have introduced and discussed the $L^\#$ algorithm. We now present a short experimental evaluation of the algorithm to demonstrate its performance when compared to other state-of-art algorithms. We run two versions of $L^\#$: the base version (Algorithm 1), and the ADS optimised variant ($L^\#_{\text{ADS}}$), and compare these with the (highly optimized) LearnLib[4] implementations of TTT, ADT,[5] and 'RS', by which we refer to L^* with Rivest-Schapire counterexample processing [52,53]. All source-code and data is available online.[6]

Implementing EquivQuery*:* We implement equivalence queries using conformance testing, which also makes output queries. We have fixed the testing tool to Hybrid-ADS[7] [57]. Hybrid-ADS has multiple configuration options, and we have set the state cover mode to "buggy", the number of extra states to check for to 10, the number of infix symbols to 10, and the mode of execution to "random", generating an infinite test-suite. Note that with these settings, the equivalence queries are not exact in general but approximated via random testing.

Data-set and metrics: We use a subset of the models available from the AutomataWiki (see [47]): we learn models for the SSH, TCP, and TLS protocols, alongside the BankCard models. The largest model in this subset has 66 states and 13 input symbols. We record the number of output queries and input symbols used during learning and testing, alongside the number of equivalence queries required to learn each model. An output query is a sequence $\sigma \in I^*$ of $|\sigma|$ input symbols and one *reset* symbol. A reset symbol returns the *system under test* (SUT) to its initial state. So *resets* denotes the number of output queries and *inputs* denotes the total number of symbols sent to the SUT. We believe that these metrics accurately portray the effort required to learn a model.

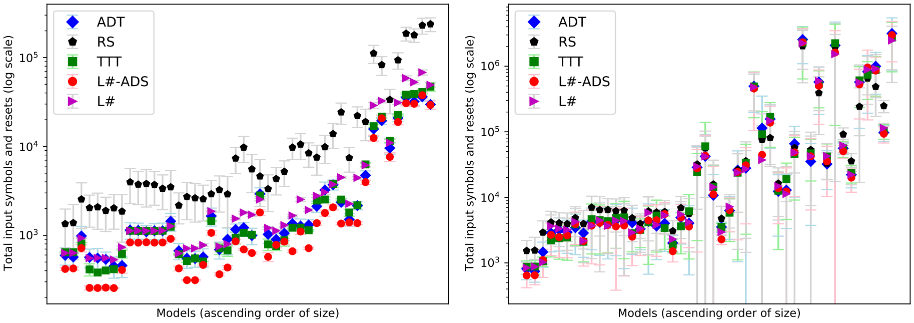
[4] https://learnlib.de/

[5] The ADT algorithm makes use of some heuristics to guide the learning process, we have selected the "Best-Effort" settings.

[6] https://gitlab.science.ru.nl/sws/lsharp and `10.5281/zenodo.5735533`

[7] https://github.com/Jaxan/hybrid-ads

Experiment Set-up: All experiments were run on a Ryzen 3700X processor with 32GB of memory, running Linux. Each experiment refers to completely learning a model of the SUT. Due to the effects of randomization in the equivalence oracle, we repeat each experiment 100 times.



(a) Symbols used during learning phase (b) Symbols used both learning and testing

Fig. 4: Performance plots of the selected learning algorithms (lower is better.)

Results and Discussion Fig. 4a shows the total size of data sent by the learning algorithms via output queries – so both the number and the size of output queries are counted. In order to incorporate the equivalence queries, Fig. 4b shows the total size of data sent to the SUT during learning and testing. Note, in both plots the y-axis is log-scaled. The x-axis indicates the models, sorted in increasing number of states. The bars indicate standard deviation.

We can observe from the learning phase plot (Fig. 4a) that $L^{\#}$ expectedly does not perform better than the TTT and ADT algorithms, while the RS algorithm performs the worst among all four. However, $L^{\#}_{\mathrm{ADS}}$ usually performs better than – or, at least, is competitive with – ADT and TTT. Furthermore, the error bars in the learning phase are very small, indicating that the measurements are stable. Generally, depending on the models a different algorithm is the fastest, but for every model, $L^{\#}_{\mathrm{ADS}}$ is among the fastest, with and without the exclusion of the testing phase.

Fig. 4b presents the total number of input symbols and resets sent to the SUT. All algorithms seem to be very close in performance, which may be explained by the testing phase dominating the process. Indeed, Aslam et al. [8] experimentally demonstrated that it is largely the testing phase which influences learning effort.

The complete benchmark results (in the appendix of [65]) show more detailed information of the learned models, and highlights the smallest number per column and model. We can see that the number of equivalence queries are roughly similar for almost all the algorithms, while $L^{\#}$ seems to perform better for some models in the learning phase.

5 Conclusions and Future Work

We presented $L^{\#}$, a new algorithm for the classical problem of active automata learning. The key idea behind the approach is to focus on establishing *apartness*, or inequivalence of states, instead of approximating equivalence as in L^* and its descendants. Concretely, the table/discrimination tree in L^*-like algorithms is replaced in $L^{\#}$ by an observation tree, together with an apartness relation. This change in perspective leads to a simple but effective algorithm, which reduces the total number of symbols required for learning when compared to state-of-the-art algorithms. In particular, the use of observation trees, which are essentially tree-shaped Mealy machines, enables a modular integration of testing techniques, such as the ADS method, to identify states. Although the asymptotic output query complexity of $L^{\#}$ is $\mathcal{O}(kn^2 + n \log m)$, in our experiments $L^{\#}$ only needs in between kn and $4kn$ output queries (resets) to learn the benchmark models (with $n \leq 66$), which means that on average $L^{\#}$ needs in between 1 and 4 output queries to identify a frontier state.

Of course there are also similarities between $L^{\#}$ and L^*. The basis of $L^{\#}$ is comparable to the top half of the L^* table: both in $L^{\#}$ and in ([53]'s version of) L^* these prefixes induce a spanning tree. The frontier of $L^{\#}$ is comparable to the bottom half of the L^* table. But whereas L^* constructs residual classes of the language, $L^{\#}$ builds an automaton directly from the observation tree. As a consequence, L^* asks redundant queries, and optimizations of L^* try to avoid this redundancy. In contrast, $L^{\#}$ does not even think about asking redundant queries since it operates directly on the observation tree and only poses queries that increase the norm.

There is still much work to do to improve our prototype implementation, to include additional conformance testing algorithms, and to extend the experimental evaluation to a richer set of benchmarks and algorithms. One issue that we need to address is scaling of $L^{\#}$ to bigger models. Our prototype implementation easily learns Mealy machines with hundreds of states, but fails to learn larger models such as the ESM benchmark of [57] (3410 states, 78 inputs) because the observation tree becomes too big ($\approx$25 million nodes will be required for the ESM). We see several ways to address this issue, e.g., pruning the observation tree, only keeping short ADSs to separate the basis states, storing parts of the tree on disk, distributing the tree over multiple processors (parallelizing the learning process), and using existing platforms for big graph processing [54].

Aslam et al. [9] report on experiments in which active learning techniques are applied to 202 industrial software components from ASML. Out of these, interface protocols could be successfully derived for 134 components (within a give time bound). One of the main conclusions of the study is that the equivalence checking phase (i.e. conformance testing of hypothesis models) is the bottleneck for scalability in industry. We believe that a tighter integration of learning and testing, as enabled by $L^{\#}$, will be key to address this challenging problem.

It will be interesting to extend $L^{\#}$ to richer frameworks such as register automata, symbolic automata and weighted automata. In fact, we discovered $L^{\#}$ while working on a grey-box learning algorithm for symbolic automata.

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample-guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) Proceedings of 18th International Symposium on Formal Methods (FM 2012). Lecture Notes in Computer Science, vol. 7436, pp. 10–27. Springer (Aug 2012). https://doi.org/10.1007/978-3-642-32759-9_4
2. Aarts, F., Vaandrager, F.: Learning I/O automata. In: Gastin, P., Laroussinie, F. (eds.) 21st International Conference on Concurrency Theory (CONCUR), 2010, Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 71–85. Springer (2010)
3. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991)
4. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers. Lecture Notes in Computer Science, vol. 11026, pp. 74–100. Springer (2018)
5. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
6. Angluin, D., Eisenstat, S., Fisman, D.: Learning regular languages via alternating automata. In: IJCAI. pp. 3308–3314. AAAI Press (2015)
7. Argyros, G., D'Antoni, L.: The learnability of symbolic automata. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018. Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 427–445. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_23
8. Aslam, K., Cleophas, L., Schiffelers, R.R.H., van den Brand, M.: Interface protocol inference to aid understanding legacy software components. Softw. Syst. Model. **19**(6), 1519–1540 (2020). https://doi.org/10.1007/s10270-020-00809-2
9. Aslam, K., Luo, Y., Schiffelers, R.R.H., van den Brand, M.: Interface protocol inference to aid understanding legacy software components. In: Hebig, R., Berger, T. (eds.) Proceedings of MODELS 2018 Workshops. CEUR Workshop Proceedings, vol. 2245, pp. 6–11. CEUR-WS.org (2018)
10. Balcázar, J.L., Díaz, J., Gavaldà, R.: Algorithms for learning finite automata from queries: A unified view. In: Du, D., Ko, K. (eds.) Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book. pp. 53–72. Kluwer (1997)
11. Balle, B., Mohri, M.: Learning weighted automata. In: CAI. Lecture Notes in Computer Science, vol. 9270, pp. 1–21. Springer (2015)
12. Barlocco, S., Kupke, C., Rot, J.: Coalgebra learning via duality. In: FoSSaCS. Lecture Notes in Computer Science, vol. 11425, pp. 62–79. Springer (2019)
13. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) Proceedings, Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005. Lecture Notes in Computer Science, vol. 3442, pp. 175–189. Springer (2005)
14. Bergadano, F., Varricchio, S.: Learning behaviors of automata from multiplicity and equivalence queries. SIAM J. Comput. **25**(6), 1268–1280 (Dec 1996). https://doi.org/10.1137/S009753979326091X

15. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Computers **21**(6), 592–597 (1972). https://doi.org/10.1109/TC.1972.5009015
16. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: IJCAI. pp. 1004–1009 (2009)
17. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016)
18. Colcombet, T., Petrisan, D., Stabile, R.: Learning automata and transducers: A categorical approach. In: CSL. LIPIcs, vol. 183, pp. 15:1–15:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
19. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (Dec 1959). https://doi.org/10.1007/BF01386390
20. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (Aug 1975). https://doi.org/10.1145/360933.360975
21. Fiterău-Broştean, P., Howar, F.: Learning-based testing the sliding window behavior of TCP implementations. *in* FMICS, LNCS **10471**, 185–200 (2017)
22. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. *in* CAV, LNCS **9780**, 454–471 (2016)
23. Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 142–151. SPIN 2017, ACM, New York, NY, USA (2017)
24. Florêncio, C.C., Verwer, S.: Regular inference as vertex coloring. Theor. Comput. Sci. **558**, 18–34 (2014). https://doi.org/10.1016/j.tcs.2014.09.023
25. Frohme, M.T.: Active automata learning with adaptive distinguishing sequences. CoRR **abs/1902.01139** (2019), http://arxiv.org/abs/1902.01139
26. Geuvers, H., Jacobs, B.: Relating apartness and bisimulation. Logical Methods in Computer Science **Volume 17, Issue 3** (Jul 2021). https://doi.org/10.46298/lmcs-17(3:15)2021
27. Groz, R., Brémond, N., da Silva Simão, A., Oriat, C.: hW-inference: A heuristic approach to retrieve models through black box testing. J. Syst. Softw. **159** (2020). https://doi.org/10.1016/j.jss.2019.110426
28. Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.D.: Efficient regression testing of CTI-systems: Testing a complex call-center solution. Annual review of communication, Int.Engineering Consortium (IEC) **55**, 1033–1040 (2001)
29. Heerdt, G.v.: CALF: Categorical Automata Learning Framework. Phd thesis, University College London (Oct 2020)
30. Heerdt, G.v., Kupke, C., Rot, J., Silva, A.: Learning weighted automata over principal ideal domains. In: Goubault-Larrecq, J., König, B. (eds.) Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020. vol. 12077, pp. 602–621. Springer (2020). https://doi.org/10.1007/978-3-030-45231-5_31
31. Heyting, A.: Zur intuitionistischen Axiomatik der projektiven Geometrie. Mathematische Annalen **98**, 491–538 (1927)
32. Howar, F.: Active learning of interface programs. Ph.D. thesis, University of Dortmund (Jun 2012)
33. Howar, F., Isberner, M., Steffen, B., Bauer, O., Jonsson, B.: Inferring semantic interfaces of data structures. In: ISoLA (1): Leveraging Applications of Formal

Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, 2012, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7609, pp. 554–571. Springer (2012)

34. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers. pp. 123–148. Springer International Publishing (2018)
35. Irfan, M.N., Oriat, C., Groz, R.: Angluin style finite state machine inference with non-optimal counterexamples. In: Proceedings of the First International Workshop on Model Inference In Testing. p. 11–19. MIIT '10, Association for Computing Machinery, New York, NY, USA (2010)
36. Isberner, M.: Foundations of active automata learning: an algorithmic perspective. Ph.D. thesis, Technical University Dortmund, Germany (2015), http://hdl.handle.net/2003/34282
37. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. pp. 307–322. Springer International Publishing, Cham (2014)
38. Kearns, M.J., Vazirani, U.V.: An introduction to computational learning theory. MIT Press (1994)
39. Lee, D., Yannakakis, M.: Testing finite-state machines: State identification and verification. IEEE Trans. Comput. **43**(3), 306–320 (1994)
40. Maler, O., Mens, I.: A generic algorithm for learning symbolic automata from membership queries. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10460, pp. 146–169. Springer (2017)
41. Maler, O., Pnueli, A.: On the learnability of infinitary regular sets. Inf. Comput. **118**(2), 316–326 (1995). https://doi.org/10.1006/inco.1995.1070
42. Margaria, T., Raffelt, H., Steffen, B.: Knowledge-based relevance filtering for efficient system-level test-based model generation. Innov. Syst. Softw. Eng. **1**(2), 147–156 (2005). https://doi.org/10.1007/s11334-005-0016-y
43. Meinke, K.: CGE: A sequential learning algorithm for Mealy automata. In: Sempere, J., García, P. (eds.) Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6339, pp. 148–162. Springer (2010)
44. Meinke, K., Niu, F., Sindhu, M.A.: Learning-based software testing: A tutorial. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011. Revised Selected Papers. Communications in Computer and Information Science, vol. 336, pp. 200–219. Springer (2011)
45. Merten, M., Howar, F., Steffen, B., Margaria, T.: Automata learning with on-the-fly direct hypothesis construction. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011. Revised Selected Papers. Communications in Computer and Information Science, vol. 336, pp. 248–260. Springer (2011)

46. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 613–625. ACM (2017). https://doi.org/10.1145/3009837.3009879
47. Neider, D., Smetsers, R., Vaandrager, F.W., Kuppens, H.: Benchmarks for automata learning and conformance testing. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 11200, pp. 390–416. Springer (2018)
48. Niese, O.: An Integrated Approach to Testing Complex Systems. Ph.D. thesis, University of Dortmund (2003)
49. Petrenko, A., Avellaneda, F., Groz, R., Oriat, C.: From passive to active FSM inference via checking sequence construction. In: Yevtushenko, N., Cavalli, A.R., Yenigün, H. (eds.) Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10533, pp. 126–141. Springer (2017)
50. Petrenko, A., Li, K., Groz, R., Hossen, K., Oriat, C.: Inferring approximated models for systems engineering. In: 15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014. pp. 249–253. IEEE Computer Society (2014). https://doi.org/10.1109/HASE.2014.46
51. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. STTT **11**(5), 393–407 (2009)
52. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences (extended abstract). In: Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, 15-17 May 1989, Seattle, Washington, USA. pp. 411–420. ACM (1989)
53. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021
54. Sakr, S., Bonifati, A., Voigt, H., Iosup, A., Ammar, K., Angles, R., Aref, W., Arenas, M., Besta, M., Boncz, P.A., Daudjee, K., Valle, E.D., Dumbrava, S., Hartig, O., Haslhofer, B., Hegeman, T., Hidders, J., Hose, K., Iamnitchi, A., Kalavri, V., Kapp, H., Martens, W., Özsu, M.T., Peukert, E., Plantikow, S., Ragab, M., Ripeanu, M.R., Salihoglu, S., Schulz, C., Selmer, P., Sequeda, J.F., Shinavier, J., Szárnyas, G., Tommasini, R., Tumeo, A., Uta, A., Varbanescu, A.L., Wu, H.Y., Yakovets, N., Yan, D., Yoneki, E.: The future is big graphs: A community view on graph processing systems. Commun. ACM **64**(9), 62–71 (Aug 2021). https://doi.org/10.1145/3434642
55. Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In: Ábrahám, E., Huisman, M. (eds.) Proceedings 12th International Conference on integrated Formal Methods (iFM). LNCS, vol. 9681, pp. 311–325 (2016)
56. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222. Springer (2009)
57. Smeenk, W., Moerman, J., Vaandrager, F.W., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, France, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9407, pp. 67–83. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_5

58. Smetsers, R., Fiterau-Brostean, P., Vaandrager, F.W.: Model learning as a satisfiability modulo theories problem. In: Klein, S.T., Martín-Vide, C., Shapira, D. (eds.) Language and Automata Theory and Applications - 12th International Conference, LATA 2018, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10792, pp. 182–194. Springer (2018)
59. Smetsers, R., Moerman, J., Jansen, D.N.: Minimal separating sequences for all pairs of states. In: Dediu, A., Janousek, J., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Proceedings. Lecture Notes in Computer Science, vol. 9618, pp. 181–193. Springer (2016). https://doi.org/10.1007/978-3-319-30000-9_14
60. Soucha, M., Bogdanov, K.: Observation tree approach: Active learning relying on testing. Comput. J. **63**(9), 1298–1310 (2020). https://doi.org/10.1093/comjnl/bxz056
61. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2 edn. (2000). https://doi.org/10.1017/CBO9781139168717
62. Urbat, H., Schröder, L.: Automata learning: An algebraic approach. In: LICS. pp. 900–914. ACM (2020)
63. Vaandrager, F.: Model learning. Communications of the ACM **60**(2), 86–95 (Feb 2017). https://doi.org/10.1145/2967606
64. Vaandrager, F., Bloem, R., Ebrahimi, M.: Learning Mealy machines with one timer. In: Leporati, A., Martín-Vide, C., Shapira, D., Zandron, C. (eds.) Language and Automata Theory and Applications - 15th International Conference, LATA 2021, Proceedings. Lecture Notes in Computer Science, vol. 12638, pp. 157–170. Springer (2021)
65. Vaandrager, F., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness (2022), https://arxiv.org/abs/2107.05419

Learning Realtime One-Counter Automata*

Véronique Bruyère[1], Guillermo A. Pérez[2], and Gaëtan Staquet[1,2](✉)

[1] University of Mons, Mons, Belgium
{veronique.bruyere,gaetan.staquet}@umons.ac.be
[2] University of Antwerp – Flanders Make, Antwerp, Belgium
guillermoalberto.perez@uantwerpen.be

Abstract. We present a new learning algorithm for realtime one-counter automata. Our algorithm uses membership and equivalence queries as in Angluin's L^* algorithm, as well as counter value queries and partial equivalence queries. In a partial equivalence query, we ask the teacher whether the language of a given finite-state automaton coincides with a counter-bounded subset of the target language. We evaluate an implementation of our algorithm on a number of random benchmarks and on a use case regarding efficient JSON-stream validation.

Keywords: Realtime one-counter automata · Active learning

1 Introduction

In *active learning*, a *learner* has to infer a model of an unknown machine by interacting with a *teacher*. Angluin's seminal L^* algorithm does precisely this for finite-state automata while using only *membership* and *equivalence queries* [4]. An important application of active learning is to learn black-box models from (legacy) software and hardware systems [17,28]. Though recent works have greatly advanced the state of the art in finite-state automata learning, handling real-world applications usually involves tailor-made abstractions to circumvent elements of the system which result in an infinite state space [1]. This highlights the need for learning algorithms that focus on more expressive models.

One-counter automata (OCAs) are obtained by extending finite-state automata with an integer-valued variable that can be increased, decreased, and tested for equality against zero. The counter allows OCAs to capture the behavior of some infinite-state systems. Additionally, their expressiveness has been shown sufficient to verify programs with lists [10] and validate XML streams [13]. To the best of our knowledge, there is no learning algorithm for general OCAs.

For *visibly* OCAs (that is, when the alphabet is such that letters determine whether the counter is decreased, increased, or not affected), Neider and Löding describe an algorithm in [27]. Besides the usual membership and equivalence queries, they use *partial equivalence queries*: given a finite-state automaton $\mathcal{A}$

* This work was partially supported by the FWO "SAILor" project (G030020N). Gaëtan Staquet is a research fellow (Aspirant) of the F.R.S– FNRS.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 244–262, 2022.
https://doi.org/10.1007/978-3-030-99524-9_13

and a bound k, does the language of $\mathcal{A}$ correspond to the k-bounded subset of the target language? Additionally, Fahmy and Roos [15] claim to have solved the case of realtime OCA (i.e., when the automaton is assumed to be configuration-deterministic and no ε-transitions are allowed). However, we were unable to understand the algorithm and proofs in that paper due to lack of precise formalization and detailed proofs. We also found an example where the provided algorithm did not produce the expected results. It is noteworthy that Böhm et al. [8] made similar remarks about related works of Roos [6,30].

Our contribution. We present a new learning algorithm for realtime one-counter automata (ROCAs). Our algorithm uses membership, equivalence and partial equivalence queries. It also makes use of *counter value queries*. That is, we make the assumption that we have an executable black box with observable counter values. We prove that our algorithm runs in exponential time and space and that it uses at most an exponential number of queries. Due to lack of space, some proofs have been omitted. We refer the interested reader to the full technical report of this work [12].

In [9], Bollig establishes a connection between OCAs with counter-value observability and visibly OCAs. We expose a similar connection and are thus able to leverage Neider and Löding's learning algorithm for visibly one-counter languages [27] as a sort of sub-routine for ours. Nevertheless, our learning algorithm is more sophisticated due to the fact that the counter values cannot be inferred from a given word. Technically, the latter required us to extend the classical definition of *observation tables* as used in, e.g., [4,27]. Entries in our tables are composed of Boolean language information as well as a counter value or a *wildcard* encoding the fact that we do not (yet) care about the value of the corresponding word. (Our use of wildcards is reminiscent of the work [25] on learning a regular language from an "inexperienced" teacher.) Moreover our tables need two sets of suffixes while only one is necessary in classical tables. Indeed we provide an example showing that making a table closed and consistent leads to an infinite loop when it has only one set of suffixes. Due to these extensions, much work is required to prove that it is always possible to make a table closed and consistent in finite time. Finally, we formulate queries for the teacher in a way which ensures the observation table eventually induces a right congruence refining the classical Myhill-Nerode congruence with counter-value information. Our computations and experiments show that the second set of suffixes is exponential leading to an exponential algorithm (instead of polynomial as in [27]).

We evaluate an implementation of our algorithm on random benchmarks and a use case inspired by [13]. Namely, we learn an ROCA model for a simple JSON schema validator — i.e., a program that verifies whether a JSON document satisfies a given JSON schema. The advantage of having a finite-state model of such a validator is that JSON-stream validation becomes trivially efficient (cf. automata-based parsing [3]).

Related work. Our assumption about counter-value observability means that the system with which we interact is a *gray box*. Several recent works make such assumptions to learn complex languages or ameliorate query-usage bounds. For instance, in [7], the authors assume they have information about the target language L in the form of a superset of it. Similarly, in [2], the authors assume L is obtained as the composition of two languages, one of which they know in advance. In [26], the teacher is assumed to have an executable automaton representation of the (infinite-word) target language and that some properties of this automaton are visible to the learner. Finally, in [16] it is assumed that constraints satisfied along the run of a system can be made visible.

2 Preliminaries

In this section we recall all necessary notions. We give a definition of realtime one-counter automaton adapted from [15,34]. We present the concept of behavior graph of such automata, inspired by the one given in [27] for visibly one-counter automata (VCAs), and state some important properties for our learning task.

An *alphabet* Σ is a non-empty finite set of *symbols*. A *word* is a finite sequence of symbols from Σ, and the *empty word* is denoted by ε. The set of all words over Σ is denoted by Σ^*. The *concatenation* of two words $u, v \in \Sigma^*$ is denoted by uv. A *language* L is a subset of Σ^*. Given a word $w \in \Sigma^*$ and a language $L \subseteq \Sigma^*$, the *set of prefixes of* w is $\mathit{Pref}(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, w = uv\}$ and the *set of prefixes of* L is $\mathit{Pref}(L) = \bigcup_{w \in L} \mathit{Pref}(w)$. Similarly, we have the sets of *suffixes* $\mathit{Suff}(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, w = vu\}$ and $\mathit{Suff}(L) = \bigcup_{w \in L} \mathit{Suff}(w)$. Moreover, L is said to be *prefix-closed* (resp. *suffix-closed*) if $L = \mathit{Pref}(L)$ (resp. $L = \mathit{Suff}(L)$). In this paper, we always work with *non-empty* languages L to avoid having to treat particular cases.

Definition 1. *A* realtime one-counter automaton (ROCA) $\mathcal{A}$ *is a tuple* $\mathcal{A} = (Q, \Sigma, \delta_{=0}, \delta_{>0}, q_0, F)$ *where: (1)* Σ *is an alphabet, (2)* Q *is a non-empty finite set of states, (3)* $q_0 \in Q$ *is the initial state, (4)* $F \subseteq Q$ *is the set of final states, and (5)* $\delta_{=0}$ *and* $\delta_{>0}$ *are two (total) transition functions defined as* $\delta_{=0} : Q \times \Sigma \to Q \times \{0, +1\}$ *and* $\delta_{>0} : Q \times \Sigma \to Q \times \{-1, 0, +1\}$.

The second component of the output of $\delta_{=0}$ and $\delta_{>0}$ gives the counter operation to apply when taking the transition. Notice that it is impossible to decrement the counter when it is already equal to zero.

A *configuration* is a pair $(q, n) \in Q \times \mathbb{N}$, that is, it contains the current state and the current counter value. The *transition relation* $\xrightarrow[\mathcal{A}]{} \subseteq (Q \times \mathbb{N}) \times \Sigma \times (Q \times \mathbb{N})$ contains $(q, n) \xrightarrow[\mathcal{A}]{a} (p, m)$ if and only $\begin{cases} \delta_{=0}(q, a) = (p, c) \land m = n + c & \text{if } n = 0, \\ \delta_{>0}(q, a) = (p, c) \land m = n + c & \text{if } n > 0. \end{cases}$
When the context is clear, we omit $\mathcal{A}$ to simply write $\xrightarrow{a}$. We lift the relation to words in the natural way. Notice that this relation is *deterministic* in the sense that given a configuration (q, n) and a word w, there exists a unique configuration (p, m) such that $(q, n) \xrightarrow{w} (p, m)$.

Given a word w, let $(q_0, 0) \xrightarrow{w} (q, n)$ be the *run* on w. When $n = 0$ and $q \in F$, we say that this run is *accepting*. The *language accepted* by $\mathcal{A}$ is the set

$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid (q_0, 0) \xrightarrow{w} (q, 0) \text{ with } q \in F\}$. If a language L is accepted by some ROCA, we say that L is a *realtime one-counter language (ROCL)*.

Given $w \in \Sigma^*$, we define the *counter value* of w according to $\mathcal{A}$, noted $c_{\mathcal{A}}(w)$, as the counter value n of the configuration (q, n) such that $(q_0, 0) \xrightarrow{w} (q, n)$. We define the *height of w according to $\mathcal{A}$*, noted $h_{\mathcal{A}}(w)$, as the maximal counter value among the prefixes of w, i.e., $h_{\mathcal{A}}(w) = \max_{x \in \mathit{Pref}(w)} c_{\mathcal{A}}(x)$.

We now introduce the concept of behavior graph of an ROCA $\mathcal{A}$, inspired from the one given for VCAs in [27]. It is a (possibly infinite) automaton based on the congruence relation $\equiv$ over Σ^* such that $u \equiv v$ if and only if for all $w \in \Sigma^*$, we have (1) $uw \in L \Leftrightarrow vw \in L$ and (2) $uw, vw \in \mathit{Pref}(L) \Rightarrow c_{\mathcal{A}}(uw) = c_{\mathcal{A}}(vw)$. The equivalence class of u is denoted by $[\![u]\!]_{\equiv}$. This relation $\equiv$ is a refinement of the Myhill-Nerode relation [18]. Its second condition depends on $\mathcal{A}$ and is limited to $\mathit{Pref}(L)$ because even if $\mathcal{A}$ has different counter values for words not in $\mathit{Pref}(L)$, we still require all those words to be equivalent.

Definition 2. *Let $\mathcal{A} = (Q, \Sigma, \delta_{=0}, \delta_{>0}, q_0, F)$ be an ROCA accepting $L \subseteq \Sigma^*$. The* behavior graph of $\mathcal{A}$ *is the automaton $BG(\mathcal{A}) = (Q_{\equiv}, \Sigma, \delta_{\equiv}, q^0_{\equiv}, F_{\equiv})$ where: (1) $Q_{\equiv} = \{[\![u]\!]_{\equiv} \mid u \in \mathit{Pref}(L)\}$ is the set of states, (2) $q^0_{\equiv} = [\![\varepsilon]\!]_{\equiv}$ is the initial state, (3) $F_{\equiv} = \{[\![u]\!]_{\equiv} \mid u \in L\}$ is the set of final states, (4) $\delta_{\equiv} : Q_{\equiv} \times \Sigma \to Q_{\equiv}$ is the transition function defined by: $\delta_{\equiv}([\![u]\!]_{\equiv}, a) = [\![ua]\!]_{\equiv}$, $\forall [\![u]\!]_{\equiv}, [\![ua]\!]_{\equiv} \in Q_{\equiv}$, $\forall a \in \Sigma$.*

Note that $Q_{\equiv} \neq \emptyset$ since $L \neq \emptyset$ by assumption. A straightforward induction shows that $BG(\mathcal{A})$ accepts $L = \mathcal{L}(\mathcal{A})$. By definition, BG_L is trim (each state is reachable and co-reachable) which implies that the transition function is partial.

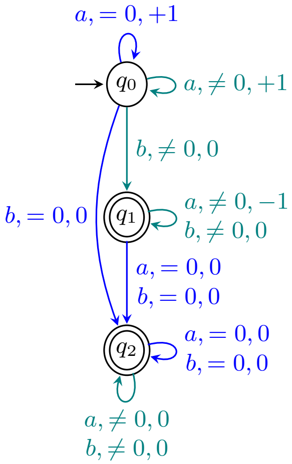


Fig. 1: An ROCA

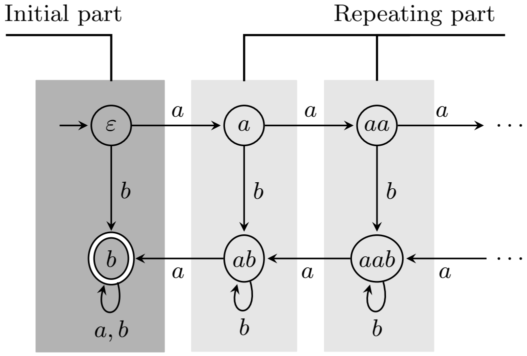


Fig. 2: Behavior graph of the left ROCA

Example 1. A 3-state ROCA $\mathcal{A}$ over $\Sigma = \{a, b\}$ is given in Figure 1. The initial state q_0 is marked by a small arrow and the two final states q_1 and q_2 are double-circled. The transitions give the input symbol, the condition on the counter value, and the counter operation, in this order ($\delta_{=0}$ is indicated in blue while $\delta_{>0}$ is indicated in green). The run on $w = aababaa$ is accepting since it ends with the configuration $(q_2, 0)$. Moreover, $c_{\mathcal{A}}(w) = 0$ and $h_{\mathcal{A}}(w) = 2$. One can verify that $\mathcal{L}(\mathcal{A}) = \{w \in \{a,b\}^* \mid \exists n \geq 0, \exists k_1, \ldots, k_n \geq 0, \exists u \in \{a,b\}^*, w = a^n b(b^{k_1} a \cdots b^{k_n} a)u\}$. The behavior graph $BG(\mathcal{A})$ of $\mathcal{A}$ is given in Figure 2. One can check that $b \equiv abba$. Indeed $\forall w \in \Sigma^*, bw \in L \Leftrightarrow abbaw \in L$. Moreover, $\forall w \in \Sigma^*$ such that $bw, abbaw \in \mathit{Pref}(L)$, we have $c_{\mathcal{A}}(bw) = c_{\mathcal{A}}(abbaw)$. However, $ab \not\equiv aab$ since $ab, aab \in \mathit{Pref}(L)$ but $c_{\mathcal{A}}(ab) = 1 \neq c_{\mathcal{A}}(aab) = 2$. □

We finally state two important properties of the behavior graph, useful for the learning of ROCAs. We first establish that $BG(\mathcal{A})$ has a finite representation that relies on the fact that it has an ultimately periodic structure (see Figure 2). Let us introduce some notations. By definition of the states of $BG(\mathcal{A})$, all words in the same class $[\![u]\!]_\equiv$ have the same counter value. We thus define the *level* ℓ of $BG(\mathcal{A})$ as the set of states with counter value ℓ. One can easily check that each level has a number of states bounded by $|Q|$. The minimal such bound is called the *width* of BG_L and is denoted by K. This observation allows to enumerate the states in level ℓ using a mapping $\nu_\ell : \{[\![w]\!]_\equiv \in Q_\equiv \mid c_{\mathcal{A}}(w) = \ell\} \to \{1, \ldots, K\}$. Using these enumerations ν_ℓ, $\ell \in \mathbb{N}$, we can encode the transitions of $BG(\mathcal{A})$ as a sequence of mappings $\tau_\ell : \{1, \ldots, K\} \times \Sigma \to \{1, \ldots, K\} \times \{-1, 0, +1\}$, with $\ell \in \mathbb{N}$, as follows. For all $i \in \{1, \ldots, K\}$, $a \in \Sigma$, the mapping τ_ℓ is defined as:

$$\tau_\ell(i, a) = \begin{cases} (j, c) & \text{if } \exists [\![u]\!]_\equiv, [\![ua]\!]_\equiv \in Q_\equiv \text{ such that } c_{\mathcal{A}}(u) = \ell, \\ & c_{\mathcal{A}}(ua) = \ell + c, \nu_\ell([\![u]\!]_\equiv) = i, \nu_{\ell+c}([\![ua]\!]_\equiv) = j, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In this way, the behavior graph can be encoded as the sequence $\alpha = \tau_0 \tau_1 \tau_2 \ldots$, called a *description* of $BG(\mathcal{A})$. The following theorem states that there always exists such a description which is *periodic* (see again Figure 2).

Theorem 1. *Let $\mathcal{A}$ be an ROCA, $BG(\mathcal{A}) = (Q_\equiv, \Sigma, \delta_\equiv, q_\equiv^0, F_\equiv)$ be the behavior graph of $\mathcal{A}$, and K be the width of $BG(\mathcal{A})$. Then, there exists a sequence of enumerations $\nu_\ell : \{[\![u]\!]_\equiv \in Q_\equiv \mid c_{\mathcal{A}}(u) = \ell\} \to \{1, \ldots, K\}$ such that the corresponding description α of $BG(\mathcal{A})$ is an ultimately periodic word with offset $m > 0$ and period $k \geq 0$, i..e, $\alpha = \tau_0 \ldots \tau_{m-1}(\tau_m \ldots \tau_{m+k-1})^\omega$.*

This theorem is the counterpart of a similar theorem given in [27] for VCAs. We get this theorem thanks to an isomorphism between the behavior graph of an ROCA $\mathcal{A}$ and that of a suitable VCA constructed from $\mathcal{A}$.

The second major property states that from a periodic description of a behavior graph $BG(\mathcal{A})$, one can construct an ROCA that accepts the same language.

Proposition 1. *Let $\mathcal{A}$ be an ROCA accepting a language $L \subseteq \Sigma^*$, $BG(\mathcal{A})$ be its behavior graph of width K, $\alpha = \tau_0 \ldots \tau_{m-1}(\tau_m \ldots \tau_{m+k-1})^\omega$ be a periodic description of $BG(\mathcal{A})$ with offset m and period k. Then, from α, one can construct an ROCA $\mathcal{A}_\alpha$ accepting L such that the size of $\mathcal{A}_\alpha$ is polynomial in m, k and K.*

3 Learning ROCAs

The aim of this paper is to design an ROCA-learning algorithm. We suppose that the reader is familiar with the concept of *active learning*, and in particular with Angluin's seminal L^* algorithm for learning finite-state automata (DFAs) [4]. In this section, let us fix a language $L \subseteq \Sigma^*$ and an ROCA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$. We here explain how a *learner* will learn L by querying a *teacher*. Our learning algorithm is inspired by the one designed in [27] for VCAs. The idea is to learn an initial fragment of the behavior graph $BG(\mathcal{A})$ up to a fixed counter limit ℓ, to extract every possible periodic description from the fragment, and to construct an ROCA from each of these descriptions. If we find one ROCA accepting L, we are done. Otherwise, we increase ℓ and repeat the process.

Formally, the initial fragment up to ℓ is called the *limited behavior graph* $BG_\ell(\mathcal{A})$. This is the subgraph of $BG(\mathcal{A})$ whose set of states is $\{[\![w]\!]_\equiv \in Q_\equiv \mid h_\mathcal{A}(w) \leq \ell\}$. This DFA accepts the language $L_\ell = \{w \in L \mid \forall x \in \mathit{Pref}(w), 0 \leq c_\mathcal{A}(x) \leq \ell\}$. Notice that $BG_\ell(\mathcal{A})$ is composed of the first $\ell + 1$ levels of $BG(\mathcal{A})$ (from 0 to ℓ) such that each level is restricted to states $[\![w]\!]_\equiv$ with $h_\mathcal{A}(w) \leq \ell$.

During the learning process, the teacher has to answer four different types of queries asked by the learner: (1) *membership query* (does $w \in \Sigma^*$ belong to L?), (2) *counter value query* (given $w \in \mathit{Pref}(L)$, what is $c_\mathcal{A}(w)$?), (3) *partial equivalence query* (does the DFA $\mathcal{B}$ accept L_ℓ?), and (4) *equivalence query* (does the ROCA $\mathcal{B}$ accept L?). In case of negative answer to a (partial) equivalence query, the teacher provides a counterexample $w \in \Sigma^*$ witness of this non-equivalence.

Recall that membership and equivalence queries are used in the L^* algorithm [4]. Additionally, partial equivalence queries are required in the VCA-learning algorithm of [27] to find the basis of a periodic description for the target automaton. However counter-value queries are not necessary because VCAs use a pushdown alphabet and the counter value can be directly inferred from the word. For general alphabets, this is no longer possible and the learner has to ask the teacher for this information. Our main result is the following theorem.

Theorem 2. *Let $\mathcal{A}$ be an ROCA accepting a language $L \subseteq \Sigma^*$. Given a teacher for L, which answers membership, counter value, and (partial) equivalence queries, an ROCA accepting L can be computed in time and space exponential in $|Q|, |\Sigma|$ and t, with Q the set of states of $\mathcal{A}$ and t the length of the longest counterexample returned by the teacher on (partial) equivalence queries. The learner asks $\mathcal{O}(t^3)$ partial equivalence queries, $\mathcal{O}(|Q|t^2)$ equivalence queries and a number of membership and counter value queries exponential in $|Q|, |\Sigma|$ and t.*

In what follows we describe the main steps of our learning algorithm. Given a counter limit ℓ, we first introduce the kind of observation table $\mathscr{O}_\ell$ we use to store the learned information about L_ℓ. Secondly, we explain what are the constraints imposed to $\mathscr{O}_\ell$ to derive a DFA $\mathcal{A}_{\mathscr{O}_\ell}$, candidate for accepting L_ℓ. Thirdly, when a counterexample is returned by the teacher to a partial equivalence query with this DFA, we explain how to update the table. Fourthly, we give the whole learning algorithm such that when $\mathcal{A}_{\mathscr{O}_\ell}$ accepts L_ℓ with ℓ big enough, a periodic description α is finally extracted from $\mathcal{A}_{\mathscr{O}_\ell}$ such that the ROCA $\mathcal{A}_\alpha$ accepts L.

Observation Table and Approximation Sets. As for learning DFAs and VCAs, we use an observation table to store the data gathered during the learning process. This table aims at approximating the equivalence classes of $\equiv$ and therefore stores information about both membership to L and counter values for words known to be in $\mathit{Pref}(L)$. It depends on a counter limit $\ell \in \mathbb{N}$ since we first want to learn $BG_\ell(\mathcal{A})$. We highlight the fact that our table uses two sets of suffixes: $\widehat{S}$ and S (contrarily to the algorithms of [4,27] that use only one set S). Intuitively, we use the set $\widehat{S}$ to store membership information and the set S for counter value information. In [27], the set S is not needed as the counter value of a word can be immediately derived from the word. This is not the case for us, as the teacher's ROCA is required to compute the counter value of a word. Therefore, we need to explicitly store that information.

Definition 3. *Let $\ell \in \mathbb{N}$ be a counter limit and L_ℓ be the language accepted by the limited behavior graph $BG_\ell(\mathcal{A})$ of an ROCA $\mathcal{A}$. An* observation table *$\mathscr{O}_\ell$ up to ℓ is a tuple $(R, S, \widehat{S}, \mathcal{L}_\ell, \mathcal{C}_\ell)$ with: (1) a finite prefix-closed set $R \subseteq \Sigma^*$ of* representatives*, (2) two finite suffix-closed sets $S, \widehat{S}$ of* separators *such that $S \subseteq \widehat{S} \subseteq \Sigma^*$, (3) a function $\mathcal{L}_\ell : (R \cup R\Sigma)\widehat{S} \to \{0, 1\}$, (4) a function $\mathcal{C}_\ell : (R \cup R\Sigma)S \to \{\bot, 0, \ldots, \ell\}$.*

Let $\mathit{Pref}(\mathscr{O}_\ell)$ be the set $\{w \in \mathit{Pref}(us) \mid u \in R \cup R\Sigma, s \in \widehat{S}, \mathcal{L}_\ell(us) = 1\}$. Then for all $u \in R \cup R\Sigma$ the following holds:

- *for all $s \in \widehat{S}$, $\mathcal{L}_\ell(us)$ is 1 if $us \in L_\ell$ and 0 otherwise,*
- *for all $s \in S$, $\mathcal{C}_\ell(us)$ is $c_\mathcal{A}(us)$ if $us \in \mathit{Pref}(\mathscr{O}_\ell)$ and $\bot$ otherwise.*

In this definition, the domains of $\mathcal{L}_\ell$ and $\mathcal{C}_\ell$ are different as already mentioned. Notice that $\mathit{Pref}(\mathscr{O}_\ell) \subseteq \mathit{Pref}(L_\ell)$. To compute the values of the table $\mathscr{O}_\ell$, the learner proceeds by asking membership and counter value queries to the teacher.

	ε	a	ba
ε	$0,0$	0	1
a	$0,1$	0	1
ab	$0,1$	1	1
aba	$1,0$	1	1
aa	$0,\bot$	0	0
b	$1,0$	1	1
abb	$0,1$	1	1
$abaa$	$1,0$	1	1
$abab$	$1,0$	1	1
aaa	$0,\bot$	0	0
aab	$0,\bot$	0	0

Fig. 3: An observation table.

	ε
ε	$0,0$
a	$0,1$
ab	$0,1$
aba	$1,0$
b	$1,0$
aa	$0,\bot$
abb	$0,\bot$
$abaa$	$1,0$
$abab$	$1,0$

	ε
ε	$0,0$
a	$0,1$
ab	$0,1$
aba	$1,0$
abb	$0,1$
b	$1,0$
aa	$0,\bot$
$abaa$	$1,0$
$abab$	$1,0$
$abba$	$1,0$
$abbb$	$0,\bot$

	ε
ε	$0,0$
a	$0,1$
ab	$0,1$
aba	$1,0$
abb	$0,1$
$abbb$	$0,1$
b	$1,0$
aa	$0,\bot$
$abaa$	$1,0$
$abab$	$1,0$
$abba$	$1,0$
$abbba$	$1,0$
$abbbb$	$0,\bot$

Fig. 4: Observation tables exposing an infinite loop when using the L^* algorithm.

Example 2. In this example, we give an observation table $\mathscr{O}_\ell$ for the ROCA $\mathcal{A}$ from Figure 1 and the counter limit $\ell = 1$, see Figure 3. Hence we want to learn $BG_\ell(\mathcal{A})$ whose set of states is given by the first two levels from Figure 2.

The first column of $\mathscr{O}_\ell$ contains the elements of $R \cup R\Sigma$ such that the upper part is constituted by $R = \{\varepsilon, a, ab, aba, aa\}$ and the lower part by $R\Sigma \setminus R$. The first row contains the elements of $\widehat{S}$ such that the left part is constituted by $S = \{\varepsilon\}$ and the right part by $\widehat{S} \setminus S$. For each element $us \in (R \cup R\Sigma)S$, we store the two values $\mathcal{L}_\ell(us)$ and $\mathcal{C}_\ell(us)$ in the intersection of row u and column s. For instance, these values are equal to $0, \perp$ for $u = aa$ and $s = \varepsilon$. For each element $us \in (R \cup R\Sigma)(\widehat{S} \setminus S)$, we have only one value $\mathcal{L}_\ell(us)$ to store. Note that $\mathit{Pref}(\mathscr{O}_\ell)$ is a proper subset of $\mathit{Pref}(L_\ell)$. For instance, $aababaa \in \mathit{Pref}(L_\ell) \setminus \mathit{Pref}(\mathscr{O}_\ell)$.

Let us now explain why it is necessary to use the additional set $\widehat{S}$ in Definition 3. Assume that we only use the set S and that the current observation table is the leftmost table $\mathscr{O}_\ell$, with $\ell = 1$, given in Figure 4 for the ROCA from Figure 1. On top of that, assume we are using the classical L^* algorithm [4]. As we can see, the table is not closed since the stored information for abb, that is, $0, \perp$, does not appear in the upper part of the table for any $u \in R$. So, we add abb in this upper part and $abba$ and $abbb$ in the lower part, to obtain the second table of Figure 4. Notice that this shift of abb has changed its stored information, which is now equal to $0, 1$. Indeed the set $\mathit{Pref}(\mathscr{O}_\ell)$ now contains abb as a prefix of $abba \in L_\ell$. Again, the new table is not closed because of $abbb$. After shifting this word in the upper part of the table, we obtain the third table of Figure 4. It is still not closed due to $abbbb$. This process will continue ad infinitum. □

To avoid an infinite loop when making the table closed, as described in the previous example, we modify both the concept of table and how to derive an equivalence relation from that table. Our solution is to introduce the set $\widehat{S}$, as already explained, but also the concept of approximation set to approximate $\equiv$.

Definition 4. *Let $\mathscr{O}_\ell = (R, S, \widehat{S}, \mathcal{L}_\ell, \mathcal{C}_\ell)$ be an observation table up to ℓ. Let $u, v \in R \cup R\Sigma$. Then, $u \in \mathit{Approx}(v)$ if and only if for all $s \in S$, we have $\mathcal{L}_\ell(us) = \mathcal{L}_\ell(vs)$ and $\mathcal{C}_\ell(us) \neq \perp \wedge \mathcal{C}_\ell(vs) \neq \perp \Rightarrow \mathcal{C}_\ell(us) = \mathcal{C}_\ell(vs)$. The set $\mathit{Approx}(v)$ is called an* approximation set*.*

In this definition, note that we consider $\perp$ values as wildcards and we focus on words with suffixes from S only (and not from $\widehat{S} \setminus S$). Interestingly, such wildcard entries in observation tables also feature in learning from an "inexperienced" teacher [25]. Just like in that work, a crucial part of our learning algorithm concerns how to obtain an equivalence relation from such an observation table (note that *Approx* does not define an equivalence relation as it is not transitive).

Example 3. Let $\mathscr{O}_\ell$ be the table from Figure 3. Let us compute $\mathit{Approx}(\varepsilon)$ (recall that we only consider $S = \{\varepsilon\}$ and not $\widehat{S} = \{a, ba\}$). We can see that $aba \notin \mathit{Approx}(\varepsilon)$ as $\mathcal{L}_\ell(aba) = 1$ and $\mathcal{L}_\ell(\varepsilon) = 0$. Moreover, $a \notin \mathit{Approx}(\varepsilon)$ since $\mathcal{C}_\ell(a) \neq \perp, \mathcal{C}_\ell(\varepsilon) \neq \perp$, and $\mathcal{C}_\ell(a) \neq \mathcal{C}_\ell(\varepsilon)$. With the same arguments, we also discard $ab, b, abb, abaa, abab$. Thus, $\mathit{Approx}(\varepsilon) = \{\varepsilon, aa, aaa, aab\}$. On the other hand, $\mathit{Approx}(aa) = \{\varepsilon, a, ab, aa, abb, aaa, aab\}$ knowing that $\mathcal{C}_\ell(aa) = \perp$. □

The following notation will be convenient later. Let $\mathcal{O}_\ell$ be an observation table and $u \in R \cup R\Sigma$. If $\mathcal{C}_\ell(u) = \bot$ (which means that $u \notin \mathit{Pref}(\mathcal{O}_\ell)$), then we say that u is a $\bot$-*word*. Let us denote by $\overline{R}$ the set R from which the $\bot$-words have been removed. We define $\overline{R \cup R\Sigma}$ in a similar way. We can now formalize the relation between $\equiv$ and *Approx*.

Proposition 2. *Let $\mathcal{O}_\ell$ be an observation table up to $\ell \in \mathbb{N}$. Then for all $u, v \in \overline{R \cup R\Sigma}$, we have $u \equiv v \Rightarrow u \in \mathit{Approx}(v)$.*

Closed and Consistent Observation Table. As for the L^* algorithm [4], we need to define the constraints a table $\mathcal{O}_\ell$ must respect in order to obtain a congruence relation from *Approx* and then to construct a DFA. This is more complex than for L^*. Namely, the table must be closed, Σ-consistent, and $\bot$-consistent. The first two constraints are close to the ones already imposed by L^*. The last one is new. Crucially, it implies that *Approx* is transitive.

Definition 5. *Let $\mathcal{O}_\ell$ be an observation table up to $\ell \in \mathbb{N}$. We say the table is:*

- closed *if* $\forall u \in R\Sigma, \mathit{Approx}(u) \cap R \neq \emptyset$,
- Σ-consistent *if* $\forall u \in R$, $\forall a \in \Sigma$,

$$ua \in \bigcap_{v \in \mathit{Approx}(u) \cap R} \mathit{Approx}(va),$$

- $\bot$-consistent *if* $\forall u, v \in R \cup R\Sigma$ *such that* $u \in \mathit{Approx}(v)$,

$$\forall s \in S,\ \mathcal{C}_\ell(us) \neq \bot \Leftrightarrow \mathcal{C}_\ell(vs) \neq \bot.$$

Example 4. Let $\mathcal{O}_\ell$ be the table from Figure 3. We have $\mathit{Approx}(b) \cap R \neq \emptyset$ because $aba \in \mathit{Approx}(b)$. More generally one can check that $\mathcal{O}_\ell$ is closed. However, $\mathcal{O}_\ell$ is not Σ-consistent. Indeed, $\varepsilon b \notin \bigcap_{v \in \mathit{Approx}(\varepsilon) \cap R} \mathit{Approx}(vb)$ since $\mathit{Approx}(\varepsilon) \cap R = \{\varepsilon, aa\}$ and $\varepsilon b \notin \mathit{Approx}(aab)$. Finally, $\mathcal{O}_\ell$ is also not $\bot$-consistent since $aa \in \mathit{Approx}(\varepsilon)$ but $\mathcal{C}_\ell(aa) = \bot$ and $\mathcal{C}_\ell(\varepsilon) = 0$. □

When $\mathcal{O}_\ell$ is closed and consistent, we define the following relation $\equiv_{\mathcal{O}_\ell}$: $\forall u, v \in R \cup R\Sigma$, $u \equiv_{\mathcal{O}_\ell} v \Leftrightarrow u \in \mathit{Approx}(v)$. This relation is a congruence over R from which we can construct a DFA $\mathcal{A}_{\mathcal{O}_\ell}$.

Definition 6. *Let $\mathcal{O}_\ell$ be a closed, Σ- and $\bot$-consistent observation table up to ℓ. From $\equiv_{\mathcal{O}_\ell}$, we define the DFA $\mathcal{A}_{\mathcal{O}_\ell} = (Q_{\mathcal{O}_\ell}, \Sigma, \delta_{\mathcal{O}_\ell}, q^0_{\mathcal{O}_\ell}, F_{\mathcal{O}_\ell})$ with: (1) $Q_{\mathcal{O}_\ell} = \{[\![u]\!]_{\equiv_{\mathcal{O}_\ell}} \mid u \in R\}$, (2) $q^0_{\mathcal{O}_\ell} = [\![\varepsilon]\!]_{\equiv_{\mathcal{O}_\ell}}$, (3) $F_{\mathcal{O}_\ell} = \{[\![u]\!]_{\equiv_{\mathcal{O}_\ell}} \mid \mathcal{L}_\ell(u) = 1\}$, and (4) the (total) transition function $\delta_{\mathcal{O}_\ell}$ is defined by $\delta_{\mathcal{O}_\ell}([\![u]\!]_{\equiv_{\mathcal{O}_\ell}}, a) = [\![ua]\!]_{\equiv_{\mathcal{O}_\ell}}$, for all $[\![u]\!]_{\equiv_{\mathcal{O}_\ell}} \in Q_{\mathcal{O}_\ell}$ and $a \in \Sigma$.*

Note that $\mathcal{A}_{\mathcal{O}_\ell}$ is consistent with the information stored in $\mathcal{O}_\ell$.

Lemma 1. *For all $u \in R \cup R\Sigma$, we have $u \in \mathcal{L}(\mathcal{A}_{\mathcal{O}_\ell}) \Leftrightarrow u \in L_\ell$.*

Making a Table Closed and Consistent. Suppose we have an observation table $\mathscr{O}_\ell$ up to ℓ and we want to make it closed, Σ- and $\bot$-consistent. We here give some intuition on how to proceed.

If the table $\mathscr{O}_\ell$ is not closed or not Σ-consistent, we proceed as in the L^* algorithm [4]. In the first case, this means that $\exists u \in R\Sigma$, $\mathit{Approx}(u) \cap R = \emptyset$. It follows that $u \notin R$ and we thus add u to R and update the table. In the second case, this means that $\exists ua \in R\Sigma, \exists v \in \mathit{Approx}(u) \cap R$, $ua \notin \mathit{Approx}(va)$. We have two cases: there exists $s \in S$ such that either $\mathcal{L}_\ell(uas) \neq \mathcal{L}_\ell(vas)$, or $\mathcal{C}_\ell(uas) \neq \bot \wedge \mathcal{C}_\ell(vas) \neq \bot \wedge \mathcal{C}_\ell(uas) \neq \mathcal{C}_\ell(vas)$. In both cases, we add as to S and to $\widehat{S}$ and we update the table.

Suppose that $\mathscr{O}_\ell$ is not $\bot$-consistent, i.e., $\exists u, v \in R \cup R\Sigma, \exists s \in S$, $u \in \mathit{Approx}(v)$ and $\mathcal{C}_\ell(us) \neq \bot \Leftrightarrow \mathcal{C}_\ell(vs) = \bot$. We call *mismatch* the latter disequality. Let us assume, without loss of generality, that $\mathcal{C}_\ell(us) \neq \bot$ and $\mathcal{C}_\ell(vs) = \bot$. So, $us \in \mathit{Pref}(\mathscr{O}_\ell)$, i.e., there exist $u' \in R \cup R\Sigma$ and $s' \in \widehat{S}$ such that $us \in \mathit{Pref}(u's')$ and $\mathcal{L}_\ell(u's') = 1$. We denote by s'' the word such that $us'' = u's'$. The idea is to add $\mathit{Suff}(s'')$ to one or both sets $S, \widehat{S}$. We have two cases:

- Suppose u' is a prefix of u. We have $s'' \in \widehat{S} \setminus S$ and add $\mathit{Suff}(s'')$ to S.
- Suppose u is a proper prefix of u'. If $vs'' \in L_\ell$ then we add $\mathit{Suff}(s'')$ to $\widehat{S}$, otherwise we add $\mathit{Suff}(s'')$ to both S and $\widehat{S}$.

The difficult task is to prove that it is always possible to make a table closed and consistent in finite time.

Proposition 3. *Given an observation table $\mathscr{O}_\ell$ up to $\ell \in \mathbb{N}$, there exists an algorithm that makes it closed, Σ- and $\bot$-consistent in a finite amount of time.*

Let us give some rough intuition. Notice that R increases only when the table is not closed, and that $S, \widehat{S}$ may increase only when the table is not consistent. Firstly, the number of times the table is not closed is bounded by the number of classes of $\equiv$ up to counter limit ℓ, by Proposition 2. Indeed, when $u \in R\Sigma \setminus R$, witness that $\mathscr{O}_\ell$ is not closed, is added to R, then it becomes the only representative in its new approximation set. Secondly, one can prove that after resolving a case where the table is not consistent, then either the size of an approximation set decreases or a mismatch is eliminated. The number of times an approximation set may decrease is bounded, because there are at most $|R \cup R\Sigma|$ distinct such sets whose size is bounded by $|R \cup R\Sigma|$. Finally, the number of mismatches to eliminate is also bounded. Hard work was necessary to get this result as, when one mismatch is eliminated when solving a case where the table is not consistent, S may increase, inducing the creation of new mismatches.

Handling Counterexamples to Partial Equivalence Queries. Let $\mathcal{A}_{\mathscr{O}_\ell}$ be the DFA constructed from a closed, Σ- and $\bot$-consistent observation table $\mathscr{O}_\ell$. If the teacher's answer to a partial equivalence query over $\mathcal{A}_{\mathscr{O}_\ell}$ is positive, then $\mathcal{A}_{\mathscr{O}_\ell}$ exactly accepts L_ℓ. Otherwise, the teacher returns a counterexample, that is, a word $w \in \Sigma^*$ such that $w \in L_\ell \Leftrightarrow w \notin \mathcal{L}(\mathcal{A}_{\mathscr{O}_\ell})$. In the latter case, we add

Pref(w) to R and update the table. We finally make the new table $\mathscr{O}'_\ell$ closed, Σ- and $\bot$-consistent. We have that $\equiv_{\mathscr{O}'_\ell}$ is a strict refinement of $\equiv_{\mathscr{O}_\ell}$.

Proposition 4. *For all $u, v \in R \cup R\Sigma$, we have $u \equiv_{\mathscr{O}'_\ell} v \Rightarrow u \equiv_{\mathscr{O}_\ell} v$. Furthermore, the index of $\equiv_{\mathscr{O}'_\ell}$ is strictly greater than the index of $\equiv_{\mathscr{O}_\ell}$.*

Since the number of classes of $\equiv$ up to counter limit ℓ is bounded by the width K and the $\ell + 1$ levels of $BG_\ell(\mathcal{A})$, by Propositions 2 to 4, we deduce that after a finite number of steps, we obtain an observation table $\mathscr{O}_\ell$ and its corresponding DFA $\mathcal{A}_{\mathscr{O}_\ell}$ such that $\mathcal{L}(\mathcal{A}_{\mathscr{O}_\ell}) = L_\ell$.

Algorithm 1 Learning an ROCA

```
Require: A teacher knowing an ROCA $\mathcal{A}$
Ensure: An ROCA accepting the same language is returned
Initialize the observation table $\mathscr{O}_\ell$ with $\ell = 0, R = S = \widehat{S} = \{\varepsilon\}$
while true do
    Make $\mathscr{O}_\ell$ closed, $\Sigma$-, and $\bot$-consistent
    Construct the DFA $\mathcal{A}_{\mathscr{O}_\ell}$ from $\mathscr{O}_\ell$
    Ask a partial equivalence query over $\mathcal{A}_{\mathscr{O}_\ell}$
    if the answer is negative then
        Update $\mathscr{O}_\ell$ with the provided counterexample        ▷ $\ell$ is not modified
    else
        Identify all periodic descriptions $\alpha_1, \dots, \alpha_n$ of $\mathcal{A}_{\mathscr{O}_\ell}$
        Construct an ROCA $\mathcal{A}_{\alpha_i}$ for each $\alpha_i$
        Ask an equivalence query over each $\mathcal{A}_{\alpha_i}$
        if the answer is true for an $\mathcal{A}_{\alpha_i}$ then return $\mathcal{A}_{\alpha_i}$
        else Select one counterexample and update $\mathscr{O}_\ell$        ▷ $\ell$ is increased
```

Learning Algorithm. We have every piece needed to give the learning algorithm for ROCAs, as presented in Algorithm 1. We initialize the observation table $\mathscr{O}_\ell$ with $\ell = 0, R = S = \widehat{S} = \{\varepsilon\}$. Then, we make the table closed, Σ-, and $\bot$-consistent, construct the DFA $\mathcal{A}_{\mathscr{O}_\ell}$, and ask for a partial equivalence query with $\mathcal{A}_{\mathscr{O}_\ell}$. If the teacher answers positively, we have learned a DFA accepting L_ℓ. Otherwise, we use the provided counterexample to update the table without increasing ℓ. Once the learned DFA $\mathcal{A}_{\mathscr{O}_\ell}$ accepts the language L_ℓ, the next proposition states that the initial fragments (up to a certain counter limit) of both $\mathcal{A}_{\mathscr{O}_\ell}$ and $BG(\mathcal{A})$ are isomorphic. This means that, once we have learned a long enough initial fragment, we can extract a periodic description from $\mathcal{A}_{\mathscr{O}_\ell}$ that is valid for $BG(\mathcal{A})$.

Proposition 5. *Let $BG(\mathcal{A})$ be the behavior graph of an ROCA $\mathcal{A}$, K be its width, and m, k be the offset and the period of a periodic description of $BG(\mathcal{A})$. Let $s = m + (K \cdot k)^4$. Let $\mathscr{O}_\ell$ be a closed, Σ- and $\bot$-consistent observation table up to $\ell > s$ such that $\mathcal{L}(\mathcal{A}_{\mathscr{O}_\ell}) = L_\ell$. Then, the trim parts of the subautomata of $BG(\mathcal{A})$ and $\mathcal{A}_{\mathscr{O}_\ell}$ restricted to the levels in $\{0, \dots, \ell - s\}$ are isomorphic.*

Hence we extract all possible periodic descriptions α from $\mathcal{A}_{\mathscr{O}_\ell}$. By Proposition 1, each description α yields an ROCA $\mathcal{A}_\alpha$ on which we ask for an equivalence query. If the teacher answers positively, we have learned an ROCA accepting L and we are done. Otherwise, we need to increase the counter limit and update the table using some of the counterexamples provided by the teacher.

Extracting every possible periodic description of $\mathcal{A}_{\mathscr{O}_\ell}$ can be performed by identifying an isomorphism between two consecutive subgraphs of $\mathcal{A}_{\mathscr{O}_\ell}$. That is, we fix values for the offset m and period k and see if the subgraphs induced by the levels m to $m + k - 1$, and by the levels $m + k$ to $m + 2k - 1$ are isomorphic (this means considering all pairs (m, k) such that $m + 2k - 1 \leq \ell$). This can be done by executing two depth-first searches in parallel [27]. Note that multiple periodic descriptions may be found, due to the finite knowledge of the learner.

In case all ROCAs constructed from $\mathcal{A}_{\mathscr{O}_\ell}$ do not accept L, we handle the counterexamples returned by the teacher as follows. If among them, there is one counterexample, say w, such that the height $h_{\mathcal{A}}(w)$ exceeds ℓ, we add $\mathit{Pref}(w)$ to R (as in the case of a negative partial equivalence query) and the new counter limit is updated to $h_{\mathcal{A}}(w)$. If none of the counterexamples have an height exceeding ℓ (this may happen due to the limited knowledge of the learner), we instead use $\mathcal{A}_{\mathscr{O}_\ell}$ directly as an ROCA and ask an equivalence query. Since $\mathcal{L}(\mathcal{A}_{\mathscr{O}_\ell}) = L_\ell$ (as the last partial equivalence query was true), the counterexample returned by the teacher necessarily has a high enough height and we proceed as above.

Complexity of the Algorithm. Let us briefly explain the complexity announced in Theorem 2 for Algorithm 1 in terms of $|Q|$ the number of states of the given ROCA and t the length of the longest counterexample returned by the teacher. The given bound on the number of (partial) equivalence queries is obtained by arguments similar to those of [27]. The number of steps in the main loop of Algorithm 1 is the (polynomial) number of partial equivalence queries. During one step in this loop, by carefully studying how we make the table closed and consistent and handle a counterexample, we get that $R \cup R\Sigma$ (resp. $\widehat{S}$) grows linearly (resp. exponentially) in $|Q|$, $|\Sigma|$, and t. We also get an exponential number of membership and equivalence queries for the whole algorithm.

4 Experiments

We evaluated our algorithm on two types of benchmarks. The first uses randomly generated ROCAs, while the second focuses on a new approach to learn an ROCA that can efficiently check if a JSON document is valid against a given JSON schema. Notice that while there exist several algorithms that infer a JSON schema from a collection of JSON documents (see survey [5]), none are based on learning techniques nor do they yield an automaton-based validation algorithm.

The ROCAs and the learning algorithm were implemented by extending the well-known Java libraries AutomataLib and LearnLib [20]. These modifications can be consulted on [31,33], while the code for the benchmarks is available on [32]. Implementation specific details (such as the libraries) are given alongside the code. The server used for the computations ran Debian 10 over Linux 5.4.73-1-pve with a 4-core Intel® Xeon® Silver 4214R Processor with 16.5M cache, and 64GB of RAM. Moreover, we used OpenJDK version 11.0.12.

4.1 Random ROCAs

We first discuss our benchmarks based on randomly generated ROCAs.

Random Generation of ROCAs. An ROCA with given size $n = |Q|$ is randomly generated such that (1) $\forall q \in Q, q$ has a probability 0.5 of being final, and (2) $\forall q \in Q, \forall a \in \Sigma, \delta_{>0}(q, a) = (p, c)$ with p a random state in Q and c a random counter operation in $\{-1, 0, +1\}$. We define $\delta_{=0}(q, a) = (p, c)$ in a similar way except that $c \in \{0, +1\}$. All random draws are assumed to come from a uniform distribution. Since this generation does not guarantee an ROCA with n reachable states, we generate 100 ROCAs and select the ROCA with a maximal number of reachable states. However, it is still possible the resulting ROCA does not have n (co)-reachable states.

Equivalence of Two ROCAs. The language equivalence problem of ROCAs is known to be decidable and NL-complete [8]. Unfortunately, the algorithm described in [8] is difficult to implement. Instead, we use an "approximate" equivalence oracle for our experiments.[3] Let $\mathcal{A}$ and $\mathcal{B}$ be two ROCAs such that $\mathcal{B}$ is the learned ROCA from a periodic description with period k. The algorithm explores the configuration space of both ROCAs in parallel. If, at some point, it reaches a pair of configurations such that one is accepting and the other not, then we have a counterexample. However, to have an algorithm that eventually stops, we need to bound the counter value of the configurations to explore. Our approach is to first explore up to counter value $|\mathcal{A} \times \mathcal{B}|^2$ (in view of [8, Proposition 18] about shortest accepting runs in an ROCA). If no counterexample is found, we add k to the bound and, with probability 0.5, a new exploration is done up to the new bound. We repeat this whole process until we find a counterexample or until the random draw forces us to stop.

Results. For our random benchmarks, we let the size $|Q|$ of the ROCA vary between one and five, and the size of $|\Sigma|$ of the alphabet between one and four. For each pair $(|Q|, |\Sigma|)$ of sizes, we execute the learning algorithm on 100 ROCAs (generated as explained above). We set a timeout of 20 minutes and a memory limit of 16GB. The number of executions with a timeout is given in Table 1 (we do not give the pairs $(|Q|, |\Sigma|)$ where every execution could finish).

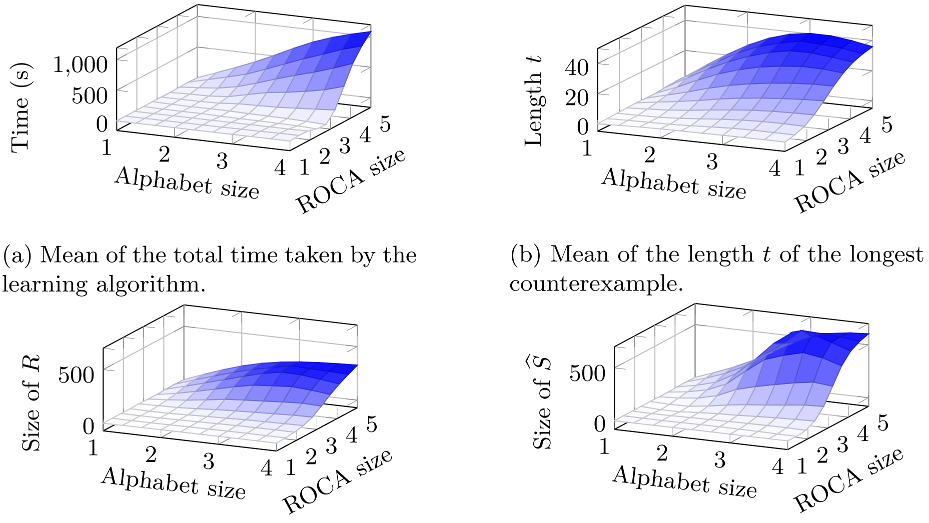
The mean of the total time taken by the algorithm is given in Figure 5a. One can see that it has an exponential growth in both sizes $|Q|$ and $|\Sigma|$. Note that

[3] The teacher might, with some small probability, answer with false positives but never with false negatives.

Table 1: Number (over 100) of executions with a timeout (TO). The executions for the missing pairs $(|Q|, |\Sigma|)$ could all finish.

$\lvert Q\rvert$	$\lvert\Sigma\rvert$	TO (20 min)
4	1	0
4	2	5
4	3	16
4	4	41
5	1	0
5	2	23
5	3	55
5	4	83

executions with a timeout had their execution time set to 20 minutes, in order to highlight the curve. Let us now drop all the executions with a timeout. The mean length of the longest counterexample provided by the teacher for (partial) equivalence queries is presented in Figure 5b and the final size of the sets R and $\widehat{S}$ is presented in Figures 5c and 5d. Note that the curves go down due to the limited number of remaining executions (for instance, the ones that could finish did not require long counterexamples). We can see that $\widehat{S}$ grows larger than R, which is coherent with the theoretical results stated at the end of Section 3.



(a) Mean of the total time taken by the learning algorithm.

(b) Mean of the length t of the longest counterexample.

(c) Mean of the final size of R.

(d) Mean of the final size of $\widehat{S}$.

Fig. 5: Results for the benchmarks based on random ROCAs.

4.2 JSON Documents and JSON Schemas

Let us now discuss the second set of benchmarks, which constitutes a proof-of-concept for an efficient validator of *JSON-document* [11] streams and is inspired by [13]. This format is currently the most popular one for exchanging information on the web. Constraints over documents can be described by a *JSON schema* [22] (like DTDs do for XML documents). See [22,21] for a brief overview of JSON documents and schemas.

In our learning process, the learner aims to construct an ROCA that can validate a JSON document, according to the schema. We assume the teacher knows the target schema and the queries are specialized as follows: (1) Membership queries: the learner provides a JSON document and the teacher answers true if the document is valid for the schema. (2) Counter value queries: the learner provides a JSON document and the teacher returns the number of unmatched `{` and `[`. Adding the two values is a heuristic abstraction that allows us to summarize two-counter information into a single counter value. Importantly, the abstraction is a design choice regarding our implementation of a teacher for these experiments and not an assumption made by our learning algorithm. (3) Partial equivalence query: the learner provides a DFA and a counter limit ℓ. The teacher randomly generates an a-priori fixed number of documents with a height not exceeding the counter limit ℓ and checks whether the DFA and the schema both agree on the documents' validity. If a disagreement is noticed, the incorrectly classified document is returned. (4) Equivalence query: the learner provides an ROCA. It is very similar to partial equivalence queries, except that documents are generated without a bound on the height. Note that the randomness of the (partial) equivalence queries implies that the learned ROCA may not completely recognize the same set of documents as for the schema.

In order for an ROCA to be learned in a reasonable time, some abstractions are made mainly to reduce the alphabet size: (1) If an object has a key named `key`, we consider the sequence of characters `"key"` as a single alphabet symbol. (2) Strings, integers, and numbers are abstracted as `"\S"`, `"\I"`, and `"\D"` respectively. Booleans are left unmodified. (3) The symbols `,`, `{`, `}`, `[`, `]`, `:` are all considered as different alphabet symbols. (4) We assume each object is composed of an ordered (instead of unordered) collection of pairs key-value. Note that the learning algorithm can learn without these restrictions but it requires substantially more time, due to a blowup in the state space or in the alphabet.

Moreover, notice that the alphabet is not known at the start of the learning process (due to the fact that keys can be any strings). Therefore we slightly modify the learning algorithm to support growing alphabets. More precisely, the learner's alphabet starts with the symbols `{` and `}` (to guarantee we can at least produce a syntactically valid JSON document for the first partial equivalence query) and is augmented each time a new symbol is seen.

Results. We considered three JSON schemas. The first is a simple document listing all possible values (i.e., it contains an integer, a double, and so on). The

Table 2: Results for JSON benchmarks.

Schema	TO (1h)	Time (s)	t	$\|R\|$	$\|\widehat{S}\|$	$\|\mathcal{A}\|$	$\|\Sigma\|$
1	0	16.39	31.00	55.55	32.00	33.00	19.00
2	27	1045.64	12.99	57.84	33.74	44.29	14.70
3	19	922.19	49.49	171.94	50.49	51.16	9.00

second is a real-world JSON schema[4] used by a code coverage tool called Codecov [14]. Finally, the third schema encodes a recursive list, i.e., an object containing a list with at most one object defined recursively. This last example is used to force the behavior graph to be infinite.

Table 2 gives the results of the benchmarks, obtained by fixing the number of random documents by (partial) equivalence query to be 1000. For each schema, 100 executions were ran with a time limit of one hour and a memory limit of 16GB by execution. We can see that real-world JSON schemas and recursively-defined schemas can be both learned by our approach. One last interesting statistics is that $|R|$ is larger than $|\widehat{S}|$, unlike for the random benchmarks.

5 Future Work

As future work, we believe one might be able to remove the use of partial equivalence queries. In this direction, perhaps replacing our use of Neider and Löding's VCA algorithm by Isberner's TTT algorithm [19] for visibly pushdown automata might help. Indeed, the TTT algorithm does not need partial equivalence queries.

Another interesting direction concerns lowering the (query) complexity of our algorithm. In [29], it is proved that L^* algorithm [4] can be modified so that adding a single separator after a failed equivalence query is enough to update the observation table. This would remove the suffix-closedness requirements on the separator sets S and $\widehat{S}$. It is not immediately clear to us whether the definition of $\perp$-consistency presented here holds in that context. Further optimizations, such as discrimination tree-based algorithms (see e.g. Kearns and Vazirani's algorithm [24]), also do not need the separator set to be suffix-closed.

It would also be interesting to directly learn the one-counter language instead of an ROCA. Indeed, our algorithm learns some ROCA that accepts the target language. It would be desirable to learn some canonical representation of the language (e.g. a minimal automaton, for some notion of minimality).

Finally, as far as we know, there currently is no active learning algorithm for deterministic one-counter automata (such that ε-transitions are allowed). We want to study how we can adapt our learning algorithm in this context.

[4] We downloaded the schema from the JSON Schema Store [23]. We modified the file to remove all constraints of type "enum".

References

1. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.W.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods Syst. Des. **46**(1), 1–41 (2015). https://doi.org/10.1007/s10703-014-0216-x, https://doi.org/10.1007/s10703-014-0216-x
2. Abel, A., Reineke, J.: Gray-box learning of serial compositions of mealy machines. In: Rayadurgam, S., Tkachuk, O. (eds.) NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9690, pp. 272–287. Springer (2016). https://doi.org/10.1007/978-3-319-40648-0_21, https://doi.org/10.1007/978-3-319-40648-0_21
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley series in computer science / World student series edition, Addison-Wesley (1986), https://www.worldcat.org/oclc/12285707
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6, https://doi.org/10.1016/0890-5401(87)90052-6
5. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C.: Schemas and types for JSON data. In: Herschel, M., Galhardas, H., Reinwald, B., Fundulaki, I., Binnig, C., Kaoudi, Z. (eds.) Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019. pp. 437–439. OpenProceedings.org (2019). https://doi.org/10.5441/002/edbt.2019.39, https://doi.org/10.5441/002/edbt.2019.39
6. Berman, P., Roos, R.: Learning one-counter languages in polynomial time (extended abstract). In: 28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987. pp. 61–67. IEEE Computer Society (1987). https://doi.org/10.1109/SFCS.1987.36, https://doi.org/10.1109/SFCS.1987.36
7. Berthon, R., Boiret, A., Pérez, G.A., Raskin, J.: Active learning of sequential transducers with side information about the domain. In: Moreira, N., Reis, R. (eds.) Developments in Language Theory - 25th International Conference, DLT 2021, Porto, Portugal, August 16-20, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12811, pp. 54–65. Springer (2021). https://doi.org/10.1007/978-3-030-81508-0_5, https://doi.org/10.1007/978-3-030-81508-0_5
8. Böhm, S., Göller, S., Jancar, P.: Bisimulation equivalence and regularity for real-time one-counter automata. J. Comput. Syst. Sci. **80**(4), 720–743 (2014). https://doi.org/10.1016/j.jcss.2013.11.003, https://doi.org/10.1016/j.jcss.2013.11.003
9. Bollig, B.: One-counter automata with counter observability. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India. LIPIcs, vol. 65, pp. 20:1–20:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.FSTTCS.2016.20, https://doi.org/10.4230/LIPIcs.FSTTCS.2016.20
10. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. Formal Methods Syst. Des. **38**(2), 158–192 (2011). https://doi.org/10.1007/s10703-011-0111-7, https://doi.org/10.1007/s10703-011-0111-7

11. Bray, T.: The javascript object notation (JSON) data interchange format. RFC **8259**, 1–16 (2017). https://doi.org/10.17487/RFC8259, https://doi.org/10.17487/RFC8259
12. Bruyère, V., Pérez, G.A., Staquet, G.: Learning realtime one-counter automata. CoRR **abs/2110.09434** (2021), https://arxiv.org/abs/2110.09434
13. Chitic, C., Rosu, D.: On validation of XML streams using finite state machines. In: Amer-Yahia, S., Gravano, L. (eds.) Proceedings of the Seventh International Workshop on the Web and Databases, WebDB 2004, June 17-18, 2004, Maison de la Chimie, Paris, France, Colocated with ACM SIGMOD/PODS 2004. pp. 85–90. ACM (2004). https://doi.org/10.1145/1017074.1017096, https://doi.org/10.1145/1017074.1017096
14. Codecov, https://about.codecov.io/
15. Fahmy, A.F., Roos, R.S.: Efficient learning of real time one-counter automata. In: Jantke, K.P., Shinohara, T., Zeugmann, T. (eds.) Algorithmic Learning Theory, 6th International Conference, ALT '95, Fukuoka, Japan, October 18-20, 1995, Proceedings. Lecture Notes in Computer Science, vol. 997, pp. 25–40. Springer (1995). https://doi.org/10.1007/3-540-60454-5_26, https://doi.org/10.1007/3-540-60454-5_26
16. Garhewal, B., Vaandrager, F.W., Howar, F., Schrijvers, T., Lenaerts, T., Smits, R.: Grey-box learning of register automata. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12546, pp. 22–40. Springer (2020). https://doi.org/10.1007/978-3-030-63461-2_2, https://doi.org/10.1007/978-3-030-63461-2_2
17. Groce, A., Peled, D.A., Yannakakis, M.: Adaptive model checking. Log. J. IGPL **14**(5), 729–744 (2006). https://doi.org/10.1093/jigpal/jzl007, https://doi.org/10.1093/jigpal/jzl007
18. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, Second Edition. Addison-Wesley (2000)
19. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_26, https://doi.org/10.1007/978-3-319-11164-3_26
20. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib - A framework for active automata learning. 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. 487–495. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_32, https://doi.org/10.1007/978-3-319-21690-4_32
21. Json.org, https://www.json.org
22. Json schema, https://json-schema.org
23. Json schema store, https://www.schemastore.org/json/
24. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press (1994), https://mitpress.mit.edu/books/introduction-computational-learning-theory
25. Leucker, M., Neider, D.: Learning minimal deterministic automata from inexperienced teachers. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering

Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7609, pp. 524–538. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_39, https://doi.org/10.1007/978-3-642-34026-0_39

26. Michaliszyn, J., Otop, J.: Learning deterministic automata on infinite words. In: Giacomo, G.D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J. (eds.) ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020). Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 2370–2377. IOS Press (2020). https://doi.org/10.3233/FAIA200367, https://doi.org/10.3233/FAIA200367
27. Neider, D., Löding, C.: Learning visibly one-counter automata in polynomial time. Tech. rep., Technical Report AIB-2010-02, RWTH Aachen (January 2010) (2010)
28. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. J. Autom. Lang. Comb. **7**(2), 225–246 (2002). https://doi.org/10.25596/jalc-2002-225, https://doi.org/10.25596/jalc-2002-225
29. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021, https://doi.org/10.1006/inco.1993.1021
30. Roos, R.S.: Deciding equivalence of deterministic one-counter automata in polynomial time with applications to learning (1988)
31. Staquet, G.: Automatalib fork for rocas, https://github.com/DocSkellington/automatalib
32. Staquet, G.: Code for the benchmarks for roca learning, https://github.com/DocSkellington/LStar-ROCA-Benchmarks
33. Staquet, G.: Learnlib fork for rocas, https://github.com/DocSkellington/Learnlib
34. Valiant, L.G., Paterson, M.: Deterministic one-counter automata. J. Comput. Syst. Sci. **10**(3), 340–350 (1975). https://doi.org/10.1016/S0022-0000(75)80005-5, https://doi.org/10.1016/S0022-0000(75)80005-5

Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic*

Ritam Raha[1,2](✉), Rajarshi Roy[3], Nathanaël Fijalkow[2,4], and Daniel Neider[3]

[1] University of Antwerp, Antwerp, Belgium
ritam.raha@uantwerpen.be
[2] CNRS, LaBRI and Université de Bordeaux, France
nathanael.fijalkow@labri.fr
[3] Max Planck Institute for Software Systems, Kaiserslautern, Germany
{rajarshi,neider}@mpi-sws.org
[4] The Alan Turing Institute of data science, United Kingdom

Abstract. Linear temporal logic (LTL) is a specification language for finite sequences (called traces) widely used in program verification, motion planning in robotics, process mining, and many other areas. We consider the problem of learning formulas in fragments of LTL without the **U**-operator for classifying traces; despite a growing interest of the research community, existing solutions suffer from two limitations: they do not scale beyond small formulas, and they may exhaust computational resources without returning any result. We introduce a new algorithm addressing both issues: our algorithm is able to construct formulas an order of magnitude larger than previous methods, and it is anytime, meaning that it in most cases successfully outputs a formula, albeit possibly not of minimal size. We evaluate the performances of our algorithm using an open source implementation against publicly available benchmarks.

Keywords: Linear Temporal Logic · Artificial Intelligence · Specification Mining

1 Introduction

Linear Temporal Logic (LTL) is a prominent logic for specifying temporal properties [20] over infinite traces, and recently introduced over finite traces [6]. In this paper, we consider finite traces but, in a small abuse of notations, call this logic LTL as well. It has become a de facto standard in many fields such as model checking, program analysis, and motion planning for robotics. Over the past five to ten years learning temporal logics (of which LTL is the core) has become an active research area and identified as an important goal in artificial intelligence:

* The first two authors contributed equally, the remaining authors are ordered alphabetically. This research was funded by the FWO G030020N project "SAILor" and Deutsche Forschungsgemeinschaft (DFG) grant no. 434592664.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 263–280, 2022.
https://doi.org/10.1007/978-3-030-99524-9_14

it formalises the difficult task of building explainable models from data. Indeed, as we will see in the examples below and as argued in the literature, e.g., by [4] and [24], LTL formulas are typically easy to interpret by human users and therefore useful as explanations. The variable free syntax of LTL and its natural inductive semantics make LTL a natural target for building classifiers separating positive from negative traces.

The fundamental problem we study here, established in [25], is to build an explainable model in the form of an LTL formula from a set of positive and negative traces. More formally (we refer to the next section for formal definitions), given a set $u_1, \ldots, u_n$ of positive traces and a set $v_1, \ldots, v_n$ of negative traces, the goal is to construct a formula φ of LTL which satisfies all u_i's and none of the v_i's. In that case, we say that φ is a separating formula or—using machine learning terminology—a classifier.

To make things concrete let us introduce our running example, a classic motion planning problem in robotics and inspired by [15]. A robot collects wastebin contents in an office-like environment and empties them in a trash container. Let us assume that there is an office o, a hallway h, a container c and a wet area w. The following are possible traces obtained in experimentation with the robot (for instance, through simulation):

$$\begin{aligned} u_1 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{h} \cdot \mathrm{c} \cdot \mathrm{h} \\ v_1 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{h} \cdot \mathrm{h} \end{aligned}$$

In LTL learning we start from these labelled data: given u_1 as positive and v_1 as negative, what is a possible classifier including u_1 but not v_1? Informally, v_1 being negative implies that the order is fixed: o must be visited before c. We look for classifiers in the form of separating formulas, for instance

$$\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}),$$

where the **F**-operator stands for "finally" and **X** for "next". Note that this formula requires to visit the office first and only then visit the container.

Assume now that two more negative traces were added:

$$\begin{aligned} v_2 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{o} \cdot \mathrm{w} \cdot \mathrm{c} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \\ v_3 &= \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{h} \cdot \mathrm{w} \cdot \mathrm{o} \cdot \mathrm{w} \cdot \mathrm{c} \cdot \mathrm{w} \cdot \mathrm{w} \end{aligned}$$

Then the previous separating formula is no longer correct, and a possible separating formula is

$$\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c}) \wedge \mathbf{G}(\neg \mathrm{w}),$$

which additionally requires the robot to never visit the wet area. Here the **G**-operator stands for "globally".

Let us emphasise at this point that for the sake of presentation, we consider only exact classifiers: a separating formula must satisfy all positive traces and none of the negative traces. However, our algorithm naturally extends to the noisy data setting where the goal is to construct an approximate classifier, replacing 'all' and 'none' by 'almost all' and 'almost none'.

State of the art. A number of different approaches have been proposed, leveraging SAT solvers [19], automata [4], and Bayesian inference [16], and extended to more expressive logics such as Property Specification Language (PSL) [24] and Computational Tree Logic (CTL) [9].

Applications include program specification [17], anomaly and fault detection [3], robotics [5], and many more: we refer to [4], Section 7, for a list of practical applications. An equivalent point of view on LTL learning is as a specification mining question. The ARSENAL [13] and FRET [14] projects construct LTL specifications from natural language, we refer to [18] for an overview.

Existing methods do not scale beyond formulas of small size, making them hard to deploy for industrial cases. A second serious limitation is that they often exhaust computational resources without returning any result. Indeed theoretical studies [11] have shown that constructing the minimal LTL formula is NP-hard already for very small fragments of LTL, explaining the difficulties found in practice.

Our approach. To address both issues, we turn to *approximation* and *anytime* algorithms. Here *approximation* means that the algorithm does not ensure minimality of the constructed formula: it does ensure that the output formula separates positive from negative traces, but it may not be the smallest one. On the other hand, an algorithm solving an optimisation problem is called *anytime* if it finds better and better solutions the longer it keeps running. In other words, anytime algorithms work by refining solutions. As we will see in the experiments, this implies that even if our algorithm timeouts it may yield some good albeit non-optimal formula.

Our algorithm targets a strict fragment of LTL, which does not contain the Until operator (nor its dual Release operator). It combines two ingredients:

- *Searching for directed formulas*: we define a space efficient dynamic programming algorithm for enumerating formulas from a fragment of LTL that we call Directed LTL.
- *Combining directed formulas*: we construct two algorithms for combining formulas using Boolean operators. The first is an off-the-shelf *decision tree algorithm*, and the second is a new greedy algorithm called *Boolean subset cover*.

The two ingredients yield two subprocedures: the first one finds directed formulas of increasing size, which are then fed to the second procedure in charge of combining them into a separating formula. This yields an anytime algorithm as both subprocedures can output separating formulas even with a low computational budget and refine them over time.

Let us illustrate the two subprocedures in our running example. The first procedure enumerates so-called *directed formulas* in increasing size; we refer to the corresponding section for a formal definition. The directed formulas $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c})$ and $\mathbf{G}(\neg \mathrm{w})$ have small size hence will be generated early on. The second procedure constructs formulas as Boolean combinations of directed formulas. Without getting into the details of the algorithms, let us note that both $\mathbf{F}(\mathrm{o} \wedge \mathbf{F}\,\mathbf{X}\,\mathrm{c})$ and $\mathbf{G}(\neg\mathrm{w})$ satisfy u_1. The first does not satisfy v_1 and the second does not satisfy v_2 and v_3. Hence their conjunction $\mathbf{F}(\mathrm{o}\wedge\mathbf{F}\,\mathbf{X}\,\mathrm{c})\wedge\mathbf{G}(\neg\mathrm{w})$ is separating, meaning it satisfies u_1 but none of v_1, v_2, v_3.

Outline. The mandatory definitions and the problem statement we deal with are described in Section 2. Section 3 shows a high-level overview of our main idea in the algorithm. The next two sections, Section 4 and Section 5 describe the two phases of our algorithm in details, in one section each. We discuss the theoretical guarantees of our algorithm in Section 6. We conclude with an empirical evaluation in Section 7.

2 Preliminaries

Traces. Let $\mathcal{P}$ be a finite set of atomic propositions. An *alphabet* is a finite non-empty set $\Sigma = 2^{\mathcal{P}}$, whose elements are called *symbols*. A finite *trace* over Σ is a finite sequence $t = a_1 a_2 \ldots a_n$ such that for every $1 \leq i \leq n$, $a_i \in \Sigma$. We say that t has length n and write $|t| = n$. For example, let $\mathcal{P} = \{p, q\}$, in the trace $t = \{p,q\} \cdot \{p\} \cdot \{q\}$ both p and q hold at the first position, only p holds in the second position, and q in the third position. Note that, throughout the paper, we only consider finite traces.

A trace is a *word* if exactly one atomic proposition holds at each position: we used words in the introduction example for simplicity, writing $h \cdot o \cdot c$ instead of $\{h\} \cdot \{o\} \cdot \{c\}$.

Given a trace $t = a_1 a_2 \ldots a_n$ and $1 \leq i \leq j \leq n$, let $t[i,j] = a_i \ldots a_j$ be the *infix* of t from position i up to and including position j. Moreover, $t[i] = a_i$ is the symbol at the i^{th} position.

Linear Temporal Logic. The syntax of Linear Temporal Logic (LTL, in short) is defined by the following grammar

$$\varphi := p \in \mathcal{P} \mid \neg p \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathbf{X}\,\varphi \mid \mathbf{F}\,\varphi \mid \mathbf{G}\,\varphi \mid \varphi\,\mathbf{U}\,\psi$$

We use the standard formulas: *true* $= p \vee \neg p$, *false* $= p \wedge \neg p$ and **last** $= \neg\,\mathbf{X}\,$*true*, which denotes the last position of the trace. As a shorthand, we use $\mathbf{X}^n\,\varphi$ for $\underbrace{\mathbf{X}\ldots\mathbf{X}}_{n \text{ times}}\varphi$.

The *size of a formula* is the size of its underlying syntax tree.

Formulas in LTL are evaluated over finite traces. To define the semantics of LTL we introduce the notation $t, i \models \varphi$, which reads 'the LTL formula φ holds over trace t from position i'. We say that t satisfies φ and we write $t \models \varphi$ when $t, 1 \models \varphi$. The definition of $\models$ is inductive on the formula φ:

- $t, i \models p \in \mathcal{P}$ if $p \in t[i]$.

- $t, i \models \mathbf{X}\,\varphi$ if $i < |t|$ and $t, i+1 \models \varphi$. It is called the ne**X**t operator.
- $t, i \models \mathbf{F}\,\varphi$ if $t, i' \models \varphi$ for some $i' \in [i, |t|]$. It is called the eventually operator (F comes from **F**inally).
- $t, i \models \mathbf{G}\,\varphi$ if $t, i' \models \varphi$ for all $i' \in [i, |t|]$. It is called the **G**lobally operator.
- $t, i \models \varphi\,\mathbf{U}\,\psi$ if $t, j \models \psi$ for some $i \leq j \leq |t|$ and $t, i' \models \varphi$ for all $i \leq i' < j$. It is called the **U**ntil operator.

The LTL Learning Problem. The LTL exact learning problem studied in this paper is the following: given a set P of positive traces and a set N of negative traces, construct a minimal LTL separating formula φ, meaning such that $t \models \varphi$ for all $t \in P$ and $t \not\models \varphi$ for all $t \in N$.

There are two relevant parameters for a sample: its *size*, which is the number of traces, and its *length*, which is the maximum length of all traces.

The problem is naturally extended to the LTL noisy learning problem where the goal is to construct an ε-separating formula, meaning such that φ satisfies all but an ε proportion of the traces in P and none but an ε proportion of the traces in N. For the sake of simplicity we present an algorithm for solving the LTL exact learning problem, and later sketch how to extend it to the noisy setting.

3 High-level view of the algorithm

Let us start with a naive algorithm for the LTL Learning Problem. We can search through all LTL formulas in some order and check whether they are separating for our sample or not. Checking whether an LTL formula is separating can be done using standard methods (for e.g. using bit vector operations [2]). However, the major drawback of this idea is that we have to search through all LTL formulas, which is hard as the number of LTL formulas grows very quickly[5].

To tackle this issue, instead of the entire LTL fragment, our algorithm (as outlined in Algorithm 1) performs an iterative search through a fragment of LTL, which we call Directed LTL (Line 4). We expand upon this in Section 4. In that section, we also describe how we can iteratively generate these Directed LTL formulas in a particular "size order" (not the usual size of an LTL formula) and evaluate these formulas over the traces in the sample efficiently using dynamic programming techniques.

To include more formulas in our search space, we generate and search through Boolean combinations of the most promising formulas of Directed LTL formulas (Line 11), which we describe in detail in Section 5. Note that, the fragment of LTL that our algorithm searches through ultimately does not include formulas with **U** operator. Thus, for readability, we use LTL to refer to the fragment LTL $\setminus$ **U** in the rest of the paper.

During the search of formulas, our algorithm searches for smaller separating formulas at each iteration than the previously found ones, if any. In fact, as a

[5] The number of LTL formulas of size k is asymptotically equivalent to $\frac{\sqrt{14}\cdot 7^k}{2\sqrt{\pi k^3}}$ [12]

Algorithm 1 Overview of our algorithm

```
1:  $B \leftarrow \emptyset$
2:  $\psi \leftarrow \emptyset$: best formula found
3:  for all $s$ in “size order” do
4:      $D \leftarrow$ all Directed LTL formulas of parameter $s$
5:      for all $\varphi \in D$ do
6:          if $\varphi$ is separating and smaller than $\psi$ then
7:              $\psi \leftarrow \varphi$
8:          end if
9:      end for
10:     $B \leftarrow B \cup D$
11:     $B \leftarrow$ Boolean combinations of the promising formulas in $B$
12:     for all $\varphi \in B$ do
13:         if $\varphi$ is separating and smaller than $\psi$ then
14:             $\psi \leftarrow \varphi$
15:         end if
16:     end for
17: end for
18: Return $\psi$
```

heuristic, once a separating formula is found, we only search through formulas that are smaller than the found separating formula. Such a heuristic, along with aiding the search for minimal formulas, also reduces the search space significantly.

Anytime property. The anytime property of our algorithm is also consequence of storing the smallest formula seen so far ((Line 7 and 14)). Once we find a separating formula, we can output it and continue the search for smaller separating formulas.

Extension to the noisy setting. The algorithm is seamlessly extended to the noisy setting by rewriting lines 6 and 13: instead of outputting only separating formulas, we output ε-separating formulas.

4 Searching for directed formulas

The first insight of our approach is the definition of a fragment of LTL that we call *directed LTL*.

A *partial symbol* is a conjunction of positive or negative atomic propositions. We write $s = p_0 \wedge p_2 \wedge \neg p_1$ for the partial symbol specifying that p_0 and p_2 hold and p_1 does not. The definition of a symbol satisfying a partial symbol is natural: for instance the symbol $\{p_0, p_2, p_4\}$ satisfies s. The *width* of a partial symbol is the number of atomic propositions it uses.

Directed LTL is defined by the following grammar:

$$\varphi = \mathbf{X}^n s \quad | \quad \mathbf{F}\,\mathbf{X}^n s \quad | \quad \mathbf{X}^n(s \wedge \varphi) \quad | \quad \mathbf{F}\,\mathbf{X}^n(s \wedge \varphi),$$

where s is a partial symbol and $n \in \{0, 1, \cdots\}$. As an example, the directed formula

$$\mathbf{F}((p \wedge q) \wedge \mathbf{F}\,\mathbf{X}^2 \neg p)$$

reads: there exists a position satisfying $p \wedge q$, and at least two positions later, there exists a position satisfying $\neg p$. The intuition behind the term "directed" is that a directed formula fixes the order in which the partial symbols occur. A non-directed formula is $\mathbf{F}\,p \wedge \mathbf{F}\,q$: there is no order between p and q. Note that Directed LTL only uses the **X** and **F** operators as well as conjunctions and atomic propositions.

Generating directed formulas. Let us consider the following problem: given the sample $S = P \cup N$, we want to generate all directed formulas together with a list of traces in S, they satisfy. Our first technical contribution and key to the scalability of our approach is an efficient solution to this problem based on dynamic programming.

Let us define a natural order in which we want to generate directed formulas. They have two parameters: *length*, which is the number of partial symbols in the directed formula, and *width*, which is the maximum of the widths of the partial symbols in the directed formula. We consider the order based on summing these two parameters:

$$(1,1), (2,1), (1,2), (3,1), (2,2), (1,3), \ldots$$

(We note that in practice, slightly more complicated orders on pairs are useful since we want to increase the length more often than the width.) Our enumeration algorithm works by generating all directed formulas of a given pair of parameters in a recursive fashion. Assuming that we already generated all directed formulas for the pair of parameters (ℓ, w), we define two procedures, one for generating the directed formulas for the parameters $(\ell + 1, w)$, and the other one for $(\ell, w + 1)$.

When we generate the directed formulas, we also keep track of which traces in the sample they satisfy by exploiting a dynamic programming table called LastPos. We define it is as follows, where φ is a directed formula and t a trace in S:

$$\textsc{LastPos}(\varphi, t) = \{i \in [1, |t|] : t[1, i] \models \varphi\}\,.$$

The main benefit of LastPos is that it meshes well with directed formulas: it is algorithmically easy to compute them recursively on the structure of directed formulas.

A useful idea is to change the representation of the set of traces S, by precomputing the lookup table Index defined as follows, where t is a trace in S, s a partial symbol, and i in $[1, |t|]$:

$$\textsc{Index}(t, s, i) = \{j \in [i + 1, |t|] : t[j] \models s\}\,.$$

The table Index can be precomputed in linear time from S, and makes the dynamic programming algorithm easier to formulate.

Having defined the important ingredients, we now present the pseudocode 2 for both increasing the length and width of a formula. For the length increase algorithm, we define two extension operators $\wedge_{=k}$ and $\wedge_{\geq k}$ that "extend" the length of a directed formula φ by including a partial symbol s in the formula. Precisely, the operator $s \wedge_{=k} \varphi$ replaces the rightmost partial symbol s' in φ with $(s' \wedge \mathbf{X}^k s)$, while $s \wedge_{\geq k} \varphi$ replaces s' with $(s' \wedge \mathbf{F}\,\mathbf{X}^k s)$. For instance, $c \wedge_{=2} \mathbf{X}(a \wedge \mathbf{X}\, b) = \mathbf{X}(a \wedge \mathbf{X}(b \wedge \mathbf{X}^2 c))$. For the width increase algorithm, we say that two directed formulas are *compatible* if they are equal except for partial symbols. For two compatible formulas, we define a *pointwise-and* ($\mathbin{\dot\wedge}$) operator that takes the conjunction of the corresponding partial symbols at the same positions. For instance, $\mathbf{X}(a \wedge \mathbf{X}\, b) \mathbin{\dot\wedge} \mathbf{X}(b \wedge \mathbf{X}\, c) = \mathbf{X}((a \wedge b) \wedge \mathbf{X}(b \wedge c))$. The actual implementation of the algorithm refines the algorithms in certain places. For instance:

- Line 3: instead of considering all partial symbols, we restrict to those appearing in at least one positive trace.
- Line 13: some computations for $\varphi_{\geq j}$ can be made redundant; a finer data structure factorises the computations.
- Line 25: using a refined data structure, we only enumerate compatible directed formulas.

Lemma 1. *Algorithm 2 generates all directed formulas and correctly computes the tables* LastPos.

The dual point of view. We use the same algorithm to produce formulas in a dual fragment to directed LTL, which uses the $\mathbf{X}$ and $\mathbf{G}$ operators, the **last** predicate, as well as disjunctions and atomic propositions. The only difference is that we swap positive and negative traces in the sample. We obtain a directed formula from such a sample and apply its negation as shown below:

$$\neg\, \mathbf{X}\, \varphi = \mathbf{last} \vee \mathbf{X} \neg\varphi \quad ; \quad \neg\, \mathbf{F}\, \varphi = \mathbf{G} \neg\varphi \quad ; \quad \neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2.$$

5 Boolean combinations of formulas

As explained in the previous section, we can efficiently generate directed formulas and dual directed formulas. Now we explain how to form a Boolean combination of these formulas in order to construct separating formulas, as illustrated in the introduction.

Boolean combination of formulas. Let us consider the following subproblem: given a set of formulas, does there exist a Boolean combination of some of the formulas that is a separating formula? We call this problem the Boolean subset

Algorithm 2 Generation of directed formulas for the set of traces S

```
procedure Search directed formulas – length increase($\ell, w$)
    for all directed formulas $\varphi$ of length $\ell$ and width $w$ do
        for all partial symbols $s$ of width at most $w$ do
            for all $t \in S$ do
                $I = \text{LastPos}(\varphi, t)$
                for all $i \in I$ do
                    $J = \text{Index}(t, s, i)$
                    for all $j \in J$ do
                        $\varphi_{=j} \leftarrow s \wedge_{=(j-i)} \varphi$
                        add $j$ to $\text{LastPos}(\varphi_{=j}, t)$
                    end for
                    for all $j' \leq \max(J)$ do
                        $\varphi_{\geq j'} \leftarrow s \wedge_{\geq(j-i)} \varphi$
                        add $J \cap [j', |t|]$ to $\text{LastPos}(\varphi_{\geq j'}, t)$
                    end for
                end for
            end for
        end for
    end for
end procedure

procedure Search directed formulas – width increase($\ell, w$)
    for all directed formulas $\varphi$ of length $\ell$ and width $w$ do
        for all directed formulas $\varphi'$ of length $\ell$ and width 1 do
            if $\varphi$ and $\varphi'$ are compatible then
                $\varphi'' \leftarrow \varphi \wedge \varphi'$
                for all $t \in S$ do
                    $\text{LastPos}(\varphi'', t) \leftarrow \text{LastPos}(\varphi, t) \cap \text{LastPos}(\varphi', t)$
                end for
            end if
        end for
    end for
end procedure
```

cover, which is illustrated in Figure 1. In this example we have three formulas φ_1, φ_2, and φ_3, each satisfying subsets of $u_1, u_2, u_3, v_1, v_2, v_3$ as represented in the drawing. Inspecting the three subsets reveals that $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ is a separating formula.

The Boolean subset cover problem is a generalization of the well known and extensively studied subset cover problem, where we are given $S_1, \ldots, S_m$ subsets of $[1, n]$, and the goal is to find a subset I of $[1, m]$ such that $\bigcup_{i \in I} S_i$ covers all of $[1, n]$ – such a set I is called a cover. Indeed, it corresponds to the case where all formulas satisfy none of the negative traces: in that case, conjunctions are not useful, and we can ignore the negative traces. The subset cover problem is known to be NP-complete. However, there exists a polynomial-time $\log(n)$-approximation algorithm called the greedy algorithm: it is guaranteed to

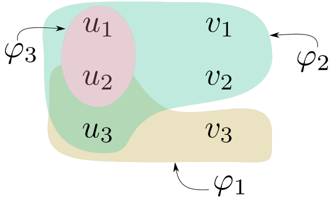


Fig. 1: The Boolean subset cover problem: the formulas φ_1, φ_2, and φ_3 satisfy the words encircled in the corresponding area; in this instance $(\varphi_1 \wedge \varphi_2) \vee \varphi_3$ is a separating formula.

construct a cover that is at most $\log(n)$ times larger than the minimum cover. This approximation ratio is optimal in the following sense [7]: there is no polynomial time $(1 - o(1)) \log(n)$-approximation algorithm for subset cover unless P = NP. Informally, the greedy algorithm for the subset cover problem does the following: it iteratively constructs a cover I by sequentially adding the most 'promising subset' to I, which is the subset S_i maximising how many more elements of $[1, n]$ are covered by adding i to I.

We introduce an extension of the greedy algorithm to the Boolean subset cover problem. The first ingredient is a scoring function, which takes into account both how close the formula is to being separating, and how large it is. We use the following score:

$$\mathit{Score}(\varphi) = \frac{\mathrm{Card}(\{t \in P : t \models \varphi\}) + \mathrm{Card}(\{t \in N : t \not\models \varphi\})}{\sqrt{|\varphi|} + 1},$$

where $|\varphi|$ is the size of φ. The use of $\sqrt{\cdot}$ is empirical, it is used to mitigate the importance of size over being separating.

The algorithm maintains a set of formulas B, which is initially the set of formulas given as input, and add new formulas to B until finding a separating formula. Let us fix a constant K, which in the implementation is set to 5. At each point in time, the algorithm chooses the K formulas $\varphi_1, \ldots, \varphi_K$ with the highest score in B and constructs all disjunctions and conjunctions of φ_i with formulas in B. For each i, we keep the disjunction or conjunction with a maximal score, and add this formula to B if it has higher score than φ_i. We repeat this procedure until we find a separating formula or no formula is added to B.

Another natural approach to the Boolean subset cover problem is to use decision trees: we use one variable for each trace and one atomic proposition for each formula to denote whether the trace satisfies the formula. We then construct a decision tree classifying all traces. We experimented with both approaches and found that the greedy algorithm is both faster and yields smaller formulas. We do not report on these experiments because the formulas output using the decision tree approach are prohibitively larger and therefore not useful for explanations. Let us, however, remark that using decision trees we get a theoretical guarantee that if there exists a separating formula as a Boolean combination of the formulas, then the algorithm will find it.

6 Theoretical guarantees

The following result shows the relevance of our approach using directed LTL and Boolean combinations.

Theorem 1. *Every formula of LTL*$(\mathbf{F}, \mathbf{X}, \wedge, \vee)$ *is equivalent to a Boolean combination of directed formulas. Equivalently, every formula of LTL*$(\mathbf{G}, \mathbf{X}, \wedge, \vee)$ *is equivalent to a Boolean combination of dual directed formulas.*

The proof of Theorem 1 can be found in the extended version of the paper [21]. To get an intuition, let us consider the formula $\mathbf{F}\, p \wedge \mathbf{F}\, q$, which is not directed, but equivalent to $\mathbf{F}(p \wedge \mathbf{F}\, q) \vee \mathbf{F}(q \wedge \mathbf{F}\, p)$. In the second formulation, there is a disjunction over the possible orderings of p and q. The formal proof generalises this rewriting idea.

This implies the following properties for our algorithm:

- *terminating*: given a bound on the size of formulas, the algorithm eventually generates all formulas of bounded size,
- *correctness*: if the algorithm outputs a formula, then it is separating,
- *completeness*: if there exists a separating formula in LTL$(\mathbf{F}, \mathbf{G}, \mathbf{X}, \wedge, \vee)$ with no nesting of $\mathbf{F}$ and $\mathbf{G}$, then the algorithm finds a separating formula.

7 Experimental evaluation

In this section, we answer the following research questions to assess the performance of our LTL learning algorithm.

RQ1: How effective are we in learning concise LTL formulas from samples?
RQ2: How much scalability do we achieve through our algorithm?
RQ3: What do we gain from the anytime property of our algorithm?

Experimental Setup. To answer the questions above, we have implemented a prototype of our algorithm in Python 3 in a tool named `SCARLET`[6] (SCalable Anytime algoRithm for LEarning lTl). We run `SCARLET` on several benchmarks generated synthetically from LTL formulas used in practice. To answer each research question precisely, we choose different sets of LTL formulas. We discuss them in detail in the corresponding sections. Note that, however, we did not consider any formulas with **U**-operator, since `SCARLET` is not designed to find such formulas.

To assess the performance of `SCARLET`, we compare it against two state-of-the-art tools for learning logic formulas from examples:

[6] https://github.com/rajarshi008/Scarlet

1. FLIE[7], developed by [19], infers minimal LTL formulas using a learning algorithm that is based on constraint solving (SAT solving).
2. SYSLITE[8], developed by [1], originally infers minimal past-time LTL formulas using an enumerative algorithm implemented in a tool called CVC4SY [23]. For our comparisons, we use a version of SYSLITE that we modified (which we refer to as SYSLITE$_{\text{L}}$) to infer LTL formulas rather than past-time LTL formulas. Our modifications include changes to the syntactic constraints generated by SYSLITE$_{\text{L}}$ as well as changing the semantics from past-time LTL to ordinary LTL.

To obtain a fair comparison against SCARLET, in both the tools, we disabled the **U**-operator. This is because if we allow **U**-operator this will only make the tools slower since they will have to search through all formulas containing **U**.

All the experiments are conducted on a single core of a Debian machine with Intel Xeon E7-8857 CPU (at 3 GHz) using up to 6 GB of RAM. We set the timeout to be 900 s for all experiments. We include scripts to reproduce all experimental results in a publicly available artifact [22].

Table 1: Common LTL formulas used in practice

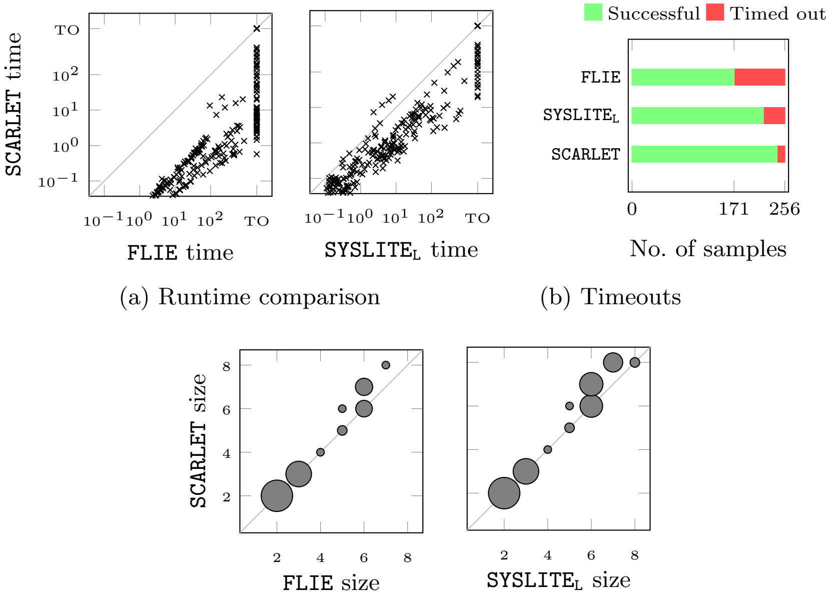
Absence:	$\mathbf{G}(\neg p)$, $\mathbf{G}(q \rightarrow \mathbf{G}(\neg p))$
Existence:	$\mathbf{F}(p)$, $\mathbf{G}(\neg p) \vee \mathbf{F}(p \wedge \mathbf{F}(q))$
Universality:	$\mathbf{G}(p)$, $\mathbf{G}(q \rightarrow \mathbf{G}(p))$
Disjunction of patterns:	$\mathbf{G}(\neg p) \vee \mathbf{F}(p \wedge \mathbf{F}(q)$ $\vee\, \mathbf{G}(\neg s) \vee \mathbf{F}(r \wedge \mathbf{F}(s))$, $\mathbf{F}(r) \vee \mathbf{F}(p) \vee \mathbf{F}(q)$

Sample generation. To provide a comparison among the learning tools, we follow the literature [19,24] and use synthetic benchmarks generated from real-world LTL formulas. For benchmark generation, earlier works rely on a fairly naive generation method. In this method, starting from a formula φ, a sample is generated by randomly drawing traces and categorizing them into positive and negative examples depending on the satisfaction with respect to φ. This method, however, often results in samples that can be separated by formulas much smaller than φ. Moreover, it often requires a prohibitively large amount of time to generate samples (for instance, for $\mathbf{G}\,p$, where almost all traces satisfy a formula) and, hence, often does not terminate in a reasonable time.

To alleviate the issues in the existing method, we have designed a novel generation method for the quick generation of large samples. In our method, we first convert the starting formula into an equivalent DFA and then extract accepted and rejected words to obtain a sample of the desired size. We provide more details on this new generation method used in the extended version [21].

[7] https://github.com/ivan-gavran/samples2LTL

[8] https://github.com/CLC-UIowa/SySLite



(a) Runtime comparison

(b) Timeouts

(c) Size comparison

Fig. 2: Comparison of SCARLET, FLIE and SYSLITE$_L$ on synthetic benchmarks. In Figure 2a, all times are in seconds and 'TO' indicates timeouts. The size of bubbles in the figure indicate the number of samples for each datapoint.

7.1 RQ1: Performance Comparison

To address our first research question, we have compared all three tools on a synthetic benchmark suite generated from eight LTL formulas. These formulas originate from a study by Dwyer et al. [8], who have collected a comprehensive set of LTL formulas arising in real-world applications (see Table 1 for an excerpt). The selected LTL formulas have, in fact, also been used by FLIE for generating its benchmarks. While FLIE also considered formulas with **U**-operator, we did not consider them for generating our benchmarks due to reasons mentioned in the experimental setup.

Our benchmark suite consists of a total of 256 samples (32 for each of the eight LTL formulas) generated using our generation method. The number of traces in the samples ranges from 50 to 2 000, while the length of traces ranges from 8 to 15.

Figure 2a presents the runtime comparison of FLIE, SYSLITE$_L$ and SCARLET on all 256 samples. From the scatter plots, we observe that SCARLET ran faster than FLIE on all samples. Likewise, SCARLET was faster than SYSLITE$_L$ on all but eight (out of 256) samples. SCARLET timed out on only 13 samples, while FLIE and SYSLITE$_L$ timed out on 85 and 36, respectively (see Figure 2b).

The good performance of SCARLET can be attributed to its efficient formula search technique. In particular, SCARLET only considers formulas that have a high potential of being a separating formula since it extracts Directed LTL formulas from the sample itself. FLIE and SYSLITE$_{\mathrm{L}}$, on the other hand, search through arbitrary formulas (in order of increasing size), each time checking if the current one separates the sample.

Figure 2c presents the comparison of the size of the formulas inferred by each tool. On 170 out of the 256 samples, all tools terminated and returned an LTL formula with size at most 7. In 150 out of this 170 samples, SCARLET, FLIE, and SYSLITE$_{\mathrm{L}}$ inferred formulas of equal size, while on the remaining 20 samples SCARLET inferred formulas that were larger. The latter observation indicates that SCARLET misses certain small, separating formulas, in particular, the ones which are not a Boolean combination of directed formulas.

However, it is important to highlight that the formulas learned by SCARLET are in most cases not significantly larger than those learned by FLIE and SYSLITE$_{\mathrm{L}}$. This can be seen from the fact that the average size of formulas inferred by SCARLET (on benchmarks in which none of the tools timed out) is 3.21, while the average size of formulas inferred by FLIE and SYSLITE$_{\mathrm{L}}$ is 3.07.

Overall, SCARLET displayed significant speed-up over both FLIE and SYSLITE$_{\mathrm{L}}$ while learning a formula similar in size, answering question RQ1 in the positive.

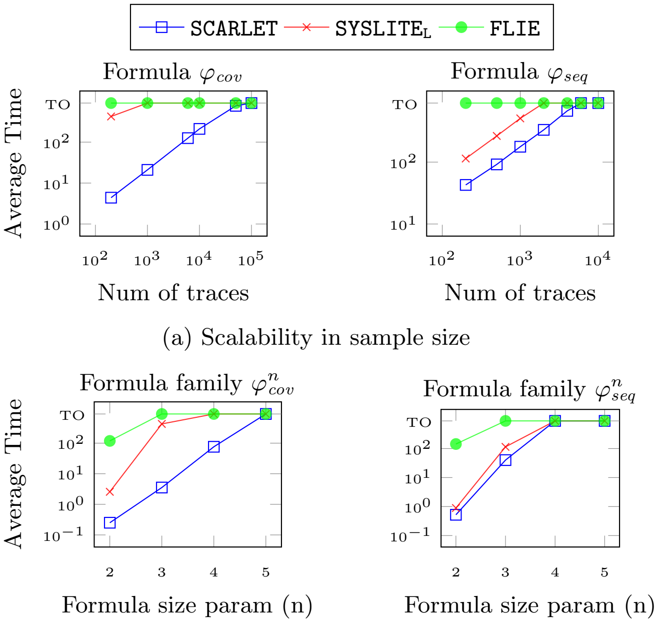
7.2 RQ2: Scalability

To address the second research question, we investigate the scalability of SCARLET in two dimensions: the size of the sample and the size of the formula from which the samples are generated.

Scalability with respect to the size of the samples. For demonstrating the scalability with respect to the size of the samples, we consider two formulas $\varphi_{cov} = \mathbf{F}(a_1) \land \mathbf{F}(a_2) \land \mathbf{F}(a_3)$ and $\varphi_{seq} = \mathbf{F}(a_1 \land \mathbf{F}(a_2 \land \mathbf{F}\, a_3))$, both of which appear commonly in robotic motion planning [10]. While the formula φ_{cov} describes the property that a robot eventually visits (or covers) three regions a_1, a_2, and a_3 in arbitrary order, the formula φ_{seq} describes that the robot has to visit the regions in the specific order $a_1 a_2 a_3$.

We have generated two sets of benchmarks for both formulas for which we varied the number of traces and their length, respectively. More precisely, the first benchmark set contains 90 samples of an increasing number of traces (5 samples for each number), ranging from 200 to 100 000, each consisting of traces of fixed length 10. On the other hand, the second benchmark set contains 90 samples of 200 traces, containing traces from length 10 to length 50. As the results on both benchmark sets are similar, we here discuss the results on the first set and refer the readers to the extended version [21] for the second set.

Figure 3a shows the average runtime results of SCARLET, FLIE, and SYSLITE$_{\mathrm{L}}$ on the first benchmark set. We observe that SCARLET substantially outperformed the other two tools on all samples. This is because both φ_{cov} and φ_{seq} are of size eight and inferring formulas of such size is computationally challenging for FLIE and SYSLITE$_{\mathrm{L}}$. In particular, FLIE and SYSLITE$_{\mathrm{L}}$ need to search through



(a) Scalability in sample size

(b) Scalability in formula size

Fig. 3: Comparison of SCARLET, FLIE and SYSLITE$_L$ on synthetic benchmarks. In Figure 3a, all times are in seconds and 'TO' indicates timeouts.

all formulas of size upto eight to infer the formulas, while, SCARLET, due to its efficient search order (using length and width of a formula), infers them faster.

From Figure 3a, we further observe a significant difference between the run times of SCARLET on samples generated from formula φ_{cov} and from formula φ_{seq}. This is evident from the fact that SCARLET failed to infer formulas for samples of φ_{seq} starting at a size of 6 000, while it could infer formulas for samples of φ_{cov} up to a size of 50 000. Such a result is again due to the search order used by SCARLET: while φ_{cov} is a Boolean combination of directed formulas of length 1 and width 1, φ_{seq} is a directed formula of length 3 and width 1.

Scalability with respect to the size of the formula. To demonstrate the scalability with respect to the size of the formula used to generate samples, we have extended φ_{cov} and φ_{seq} to families of formulas $(\varphi^n_{cov})_{n\in\mathbb{N}\setminus\{0\}}$ with $\varphi^n_{cov} = \mathbf{F}(a_1) \wedge \mathbf{F}(a_2) \wedge \ldots \wedge \mathbf{F}(a_n)$ and $(\varphi^n_{seq})_{n\in\mathbb{N}\setminus\{0\}}$ with $\varphi^n_{seq} = \mathbf{F}(a_1 \wedge \mathbf{F}(a_2 \wedge \mathbf{F}(\ldots \wedge \mathbf{F}\, a_n)))$, respectively. These family of formulas describe properties similar to that of φ_{cov} and φ_{seq}, but the number of regions is parameterized by $n \in \mathbb{N} \setminus \{0\}$. We consider formulas from the two families by varying n from 2 to 5 to generate a benchmark suite consisting of samples (5 samples for each formula) having 200 traces of length 10.

Figure 3b shows the average run time comparison of the tools for samples from increasing formula sizes. We observe a trend similar to Figure 3a: SCARLET performs better than the other two tools and infers formulas of family φ^n_{cov} faster than that of φ^n_{seq}. However, contrary to the near linear increase of the runtime with the number of traces, we notice an almost exponential increase of the runtime with the formula size.

Overall, our experiments show better scalability with respect to sample and formula size compared against the other tools, answering RQ2 in the positive.

7.3 RQ3: Anytime Property

To answer RQ3, we list two advantages of the anytime property of our algorithm. We demonstrate these advantages by showing evidence from the runs of `SCARLET` on benchmarks used in RQ1 and RQ2.

First, in the instance of a time out, our algorithm may find a "concise" separating formula while the other tools will not. In our experiments, we observed that for all benchmarks used in RQ1 and RQ2, `SCARLET` obtained a formula even when it timed out. In fact, in the samples from φ^5_{cov} used in RQ2, `SCARLET` (see Figure 3b) obtained the exact original formula, that too within one second (0.7 seconds in average), although timed out later. The time out was because `SCARLET` continued to search for smaller formulas even after finding the original formula.

Second, our algorithm can actually output the final formula earlier than its termination. This is evident from the fact that, for the 243 samples in RQ1 where `SCARLET` does not time out, the average time required to find the final formula is 10.8 seconds, while the average termination time is 25.17 seconds. Thus, there is a chance that even if one stops the algorithm earlier than its termination, one can still obtain the final formula.

Our observations from the experiments clearly indicate the advantages of anytime property to obtain a concise separating formula and thus, answering RQ3 in the positive.

8 Conclusion

We have proposed a new approach for learning temporal properties from examples, fleshing it out in an approximation anytime algorithm. We have shown in experiments that our algorithm outperforms existing tools in two ways: it scales to larger formulas and input samples, and even when it timeouts it often outputs a separating formula.

Our algorithm targets a strict fragment of LTL, restricting its expressivity in two aspects: it does not include the **U** ("until") operator, and we cannot nest the eventually and globally operators. We leave for future work to extend our algorithm to full LTL.

An important open question concerns the theoretical guarantees offered by the greedy algorithm for the Boolean subset cover problem. It extends a well known algorithm for the classic subset cover problem and this restriction has been proved to yield an optimal $\log(n)$-approximation. Do we have similar guarantees in our more general setting?

References

1. Arif, M.F., Larraz, D., Echeverria, M., Reynolds, A., Chowdhury, O., Tinelli, C.: SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In: Formal Methods in Computer Aided Design, FMCAD (2020)
2. Baresi, L., Kallehbasti, M.M.P., Rossi, M.: Efficient scalable verification of LTL specifications. In: ICSE (1). pp. 711–721. IEEE Computer Society (2015)
3. Bombara, G., Vasile, C.I., Penedo Alvarez, F., Yasuoka, H., Belta, C.: A Decision Tree Approach to Data Classification using Signal Temporal Logic. In: Hybrid Systems: Computation and Control, HSCC (2016). https://doi.org/10.1145/2883817.2883843
4. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. International Conference on Automated Planning and Scheduling, ICAPS (2019), https://ojs.aaai.org/index.php/ICAPS/article/view/3529
5. Chou, G., Ozay, N., Berenson, D.: Explaining multi-stage tasks by learning temporal logic formulas from suboptimal demonstrations. In: Robotics: Science and Systems (2020). https://doi.org/10.15607/RSS.2020.XVI.097
6. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. p. 854–860. IJCAI '13, AAAI Press (2013)
7. Dinur, I., Steurer, D.: Analytical approach to parallel repetition. In: Symposium on Theory of Computing, STOC. pp. 624–633 (2014). https://doi.org/10.1145/2591796.2591884
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering, ICSE (1999). https://doi.org/10.1145/302405.302672
9. Ehlers, R., Gavran, I., Neider, D.: Learning properties in LTL ∩ ACTL from positive examples only. In: Formal Methods in Computer Aided Design, FMCAD. pp. 104–112 (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_17
10. Fainekos, G.E., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for mobile robots. In: International Conference on Robotics and Automation, ICRA (2005). https://doi.org/10.1109/ROBOT.2005.1570410
11. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: International Conference on Grammatical Inference, ICGI (2021), https://proceedings.mlr.press/v153/fijalkow21a.html
12. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
13. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: automatic requirements specification extraction from natural language. In: NASA Formal Methods, NFM (2016). https://doi.org/10.1007/978-3-319-40648-0_4
14. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ (2020), http://ceur-ws.org/Vol-2584/PT-paper4.pdf
15. Grover, K., Barbosa, F.S., Tumova, J., Kretínský, J.: Semantic abstraction-guided motion planning for scltl missions in unknown environments. In: Robotics: Science and Systems XVII (2021). https://doi.org/10.15607/RSS.2021.XVII.090
16. Kim, J., Muise, C., Shah, A., Agarwal, S., Shah, J.: Bayesian inference of linear temporal logic specifications for contrastive explanations. In: International Joint Conference on Artificial Intelligence, IJCAI (2019). https://doi.org/10.24963/ijcai.2019/776

17. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: International Conference on Automated Software Engineering, ASE (2015). https://doi.org/10.1109/ASE.2015.71
18. Li, W.: Specification Mining: New Formalisms, Algorithms and Applications. Ph.D. thesis, University of California, Berkeley, USA (2013), http://www.escholarship.org/uc/item/4027r49r
19. Neider, D., Gavran, I.: Learning linear temporal properties. In: Formal Methods in Computer Aided Design, FMCAD (2018). https://doi.org/10.23919/FMCAD.2018.8603016
20. Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science, SFCS (1977). https://doi.org/10.1109/SFCS.1977.32
21. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning formulas in linear temporal logic. CoRR **abs/2110.06726** (2021), https://arxiv.org/abs/2110.06726
22. Raha, R., Roy, R., Fijalkow, N., Neider, D.: SCARLET: Scalable Anytime Algorithm for Learning LTL (Jan 2022). https://doi.org/10.5281/zenodo.5890149, https://doi.org/10.5281/zenodo.5890149
23. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Computer-Aided Verification, CAV (2019). https://doi.org/10.1007/978-3-030-25543-5_5
24. Roy, R., Fisman, D., Neider, D.: Learning interpretable models in the property specification language. In: International Joint Conference on Artificial Intelligence, IJCAI. pp. 2213–2219 (2020). https://doi.org/10.24963/ijcai.2020/306
25. Rozier, K.Y.: Specification: The biggest bottleneck in formal methods and autonomy. In: Verified Software. Theories, Tools, and Experiments, VSTTE (2016). https://doi.org/10.1007/978-3-319-48869-1_2

Learning Model Checking and the Kernel Trick for Signal Temporal Logic on Stochastic Processes*

Luca Bortolussi[1,2](✉), Giuseppe Maria Gallo[1], Jan Křetínský(✉)[3], and Laura Nenzi[1,4](✉)

[1] University of Trieste, Italy
[2] Modelling and Simulation Group, Saarland University, Germany
[3] Technical University of Munich, Germany
[4] University of Technology, Vienna, Austria

Abstract. We introduce a similarity function on formulae of signal temporal logic (STL). It comes in the form of a *kernel function*, well known in machine learning as a conceptually and computationally efficient tool. The corresponding *kernel trick* allows us to circumvent the complicated process of feature extraction, i.e. the (typically manual) effort to identify the decisive properties of formulae so that learning can be applied. We demonstrate this consequence and its advantages on the task of *predicting (quantitative) satisfaction* of STL formulae on stochastic processes: Using our kernel and the kernel trick, we learn (i) computationally efficiently (ii) a practically precise predictor of satisfaction, (iii) avoiding the difficult task of finding a way to explicitly turn formulae into vectors of numbers in a sensible way. We back the high precision we have achieved in the experiments by a theoretically sound PAC guarantee, ensuring our procedure efficiently delivers a close-to-optimal predictor.

1 Introduction

Is it possible to predict the probability that a system satisfies a property without knowing or executing *the system, solely based on previous experience with the system behaviour w.r.t. some* other *properties? More precisely, let $\mathbb{P}_M[\varphi]$ denote the probability that a (linear-time) property φ holds on a run of a stochastic process M. Is it possible to predict $\mathbb{P}_M[\varphi]$ knowing only $\mathbb{P}_M[\psi_i]$ for properties $\psi_1, \ldots, \psi_k$, which were randomly chosen (a-priori, not knowing φ) and thus do not necessarily have any logical relationship, e.g. implication, to φ?*

While this question cannot be in general answered with complete reliability, we show that in the setting of signal temporal logic, under very mild assumptions, it can be answered with high accuracy and low computational costs.

* This research was funded in part by the German Research Foundation (DFG) project No. 383882557 "Statistical Unbounded Verification" (KR 4890/2-1), by the Austrian FWF projects ZK-35 and by the Italian PRIN project "SEDUCE" n. 2017TWR-CNB. Corresponding authors: `lbortolussi@units.it`, `jan.kretinsky@tum.de`, `lnenzi@units.it`

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 281–300, 2022.
https://doi.org/10.1007/978-3-030-99524-9_15

Probabilistic verification and its limits. Stochastic processes form a natural way of capturing systems whose future behaviour is determined at each moment by a unique (but possibly unknown) probability measure over the successor states. The vast range of applications includes not only engineered systems such as software with probabilistic instructions or cyber-physical systems with failures but also naturally occurring systems such as biological systems. In all these cases, predictions of the system behaviour may be required even in cases the system is not (fully) known or is too large. For example, consider a safety-critical cyber-physical system with a third-party component, or a complex signalling pathway to be understood and medically exploited.

Probabilistic model checking, e.g. [4], provides a wide repertoire of analysis techniques, in particular to determine the probability $\mathbb{P}_M[\varphi]$ that the system M satisfies the logical formula φ. However, there are two caveats. Firstly, despite recent advances, [12] the scalability is still quite limited, compared to e.g. hardware or software verification. Moreover, this is still the case even if we only require *approximate* answers, i.e., for a given precision ε, to determine v such that $\mathbb{P}_M[\varphi] \in [v - \varepsilon, v + \varepsilon]$. Secondly, knowledge of the model M is required to perform the analysis.

Statistical model checking [33] fights these two issues at an often acceptable cost of relaxing the guarantee to *probably approximately* correct (PAC), requiring that the approximate answer of the analysis may be incorrect with probability at most δ. This allows for a statistical evaluation: Instead of analyzing the model, we evaluate the satisfaction of the given formula on a number of observed runs of the system and derive a statistical prediction, which is valid only with some confidence. Nevertheless, although M may be unknown, it is still necessary to execute the system in order to obtain its runs.

"Learning" model checking is a new paradigm we propose, in order to fill in a hole in the model-checking landscape where very little access to the system is possible. We are given a set of input-output pairs for model checking, i.e., a collection $\{(\psi_i, p_i)\}_i$ of formulae and their satisfaction values on a given model M, where p_i can be the probability $\mathbb{P}_M[\psi_i]$ of satisfying ψ_i, or its robustness (in case of real-valued logics), or any other quantity. From the data, we learn a predictor for the model checking problem: a classifier for Boolean satisfaction, or a regressor for quantitative domains of p_i. Note that apart from the results on the a-priori given formulae, no knowledge of the system is required; also, no runs are generated and none have to be known. As an example consequence, a user can investigate properties of a system even before buying it, solely based on producer's guarantees on the standardized formulae ψ_i.

Advantages of our approach can be highlighted as follows, not intending to replace standard model checking in standard situations but focusing on the case of extremely limited (i) information and (ii) online resources. *Probabilistic* model checking re-analyzes the system for every new property on the input; *statistical* model checking can generate runs and then, for every new property, analyzes these runs; *learning* model checking performs one analysis with complexity dependent only on the size of the data set (a-priori formulae) and then, for every new formula on input, only evaluates a simple function (whose size is again independent of the system and the property, and depends only on the data set size). Consequently, it has the least access to information and the least computational demands. While lack of any guarantees is typical for machine-learning techniques and, in this context with the lowest resources required, expectable, yet we provide PAC guarantees.

Technique and our approach. To this end, we show how to efficiently learn on the space of temporal formulae via the so-called *kernel trick*, e.g. [32]. This in turn requires to introduce a mapping of formulae to vectors (in a Hilbert space) that preserves the information on the formulae. *How to transform a formula into a vector of numbers (of always the same length)?* While this is not clear at all for finite vectors, we take the dual perspective on formulae, namely as functionals mapping trajectories to values. This point of view provides us with a large bag of functional analysis tools [11] and allows us to define the needed semantic similarity of two formulae (the inner product on the Hilbert space).

Application examples. Having discussed the possibility of learning model checking, the main potential of our kernel (and generally introducing kernels for any further temporal logics) is that it opens the door to *efficient learning on formulae* via kernel-based machine-learning techniques [27,31]. Let us sketch a few further applications that immediately suggest themselves:

Game-based synthesis Synthesis with temporal-logic specifications can often be solved via games on graphs [25,19]. However, exploration of the game graph and finding a winning strategy is done by graph algorithms ignoring the logical information. For instance, choosing between a and $\neg a$ is tried out blindly even for specifications that require us to visit as. Approaches such as [21] demonstrate how to tackle this but hit the barrier of inefficient learning of formulae. Our kernel will allow for learning reasonable choices from previously solved games.

Translating, sanitizing and simplifying specifications A formal specification given by engineers might be somewhat different from their actual intention. Using the kernel, we can, for instance, find the closest simple formula to their inadequate translation from English to logic, which is then likely to match better. (Moreover, the translation would be easier to automate by natural language processing since learning from previous cases is easy once the kernel gives us an efficient representation for formulae learning.)

Requirement mining A topic which received a lot of attention recently is that of identifying specifications from observed data, i.e. to tightly characterize a set of observed behaviours or anomalies [7]. Typical methods are using either formulae templates [6] or methods based e.g. on decision trees [9] or genetic algorithms [28]. Our kernel opens a different strategy to tackle this problem: lifting the search problem from the discrete combinatorial space of syntactic structures of formulae to a continuous space in which distances preserve semantic similarity (using e.g. kernel PCA [27] to build finite-dimensional embeddings of formulae into $\mathbb{R}^m$).

Our main contributions are the following:

- From the technical perspective, we define a kernel function for temporal formulae (of signal temporal logic, see below) and design an efficient way to learn it. This includes several non-standard design choices, improving the quality of the predictor (see Conclusions).
- Thereby we open the door to various learning-based approaches for analysis and synthesis and further applications, in particular also to what we call the *learning* model checking.
- We demonstrate the efficiency practically on predicting the expected satisfaction of formulae on stochastic systems. We complement the experimental results with a theoretical analysis and provide a PAC bound.

1.1 Related Work

Signal temporal logic (STL) [24] is gaining momentum as a requirement specification language for complex systems and, in particular, cyber-physical systems [7]. STL has been applied in several flavours, from runtime-monitoring [7], falsification problems [17] to control synthesis [18], and recently also within learning algorithms, trying to find a maximally discriminating formula between sets of trajectories [9,6]. In these applications, a central role is played by the real-valued quantitative semantics [15], measuring robustness of satisfaction. Most of the applications of STL have been directed to deterministic (hybrid) systems, with less emphasis on non-deterministic or stochastic ones [5].

Metrics and distances form another area in which formal methods are providing interesting tools, in particular logic-based distances between models, like bisimulation metrics for Markov models [2,3,1], which are typically based on a branching logic. In fact, extending these ideas to linear time logic is hard [14], and typically requires statistical approximations. Finally, another relevant problem is how to measure the distance between two logic formulae, thus giving a metric structure to the formula space, a task relevant for learning which received little attention for STL, with the notable exception of [23].

Kernels make it possible to work in a *feature space* of a higher dimension without increasing the computational cost. Feature space, as used in machine learning [31,13], refers to an n-dimensional real space that is the co-domain of a mapping from the original space of data. The idea is to map the original space in a new one that is easier to work with. The so-called *kernel trick*, e.g. [32] allows us to efficiently perform approximation and learning tasks over the feature space without explicitly constructing it. We provide the necessary background information in Section 2.2.

Overview of the paper: Section 2 recalls STL and the classic kernel trick. Section 3 provides an overview of our technique and results. Section 4 then discusses all the technical development in detail. In Section 5, we experimentally evaluate the accuracy of our learning method. In Section 6, we conclude with future work.

2 Background

Let $\mathbb{R}, \mathbb{R}_{\geq 0}, \mathbb{Q}, \mathbb{N}$ denote the sets of non-negative real, rational, and (positive) natural numbers, respectively. For vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ (with $n \in \mathbb{N}$), we write $\boldsymbol{x} = (x_1, \ldots, x_n)$ to access the components of the vectors, in contrast to sequences of vectors $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots \in \mathbb{R}^n$. Further, we write $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=1}^{n} x_i y_i$ for the scalar product of vectors.

2.1 Signal Temporal Logic

Signal Temporal Logic (STL) [24] is a linear-time temporal logic suitable to monitor properties of trajectories. A *trajectory* is a function $\xi : I \rightarrow D$ with a *time domain* $I \subseteq \mathbb{R}_{\geq 0}$, and a *state space* $D \subseteq \mathbb{R}^n$ for some $n \in \mathbb{N}$. We define the *trajectory space* $\mathcal{T}$ as the set of all possible continuous functions[5] over D. An *atomic predicate* of STL is a continuous computable predicate[6] on $\boldsymbol{x} \in \mathbb{R}^n$ of the form of $f(x_1, ..., x_n) \geq 0$, typically linear, i.e. $\sum_{i=1}^{n} q_i x_i \geq 0$ for $q_1, \ldots, q_n \in \mathbb{Q}$.

Syntax. The set $\mathcal{P}$ of STL formulae is given by the following syntax:

$$\varphi := tt \mid \pi \mid \neg\, \varphi \mid \varphi_1 \,\wedge\, \varphi_2 \mid \varphi_1 \mathbf{U}_{[a,b]} \varphi_2$$

where tt is the Boolean *true* constant, π ranges over atomic predicates, *negation* $\neg$ and *conjunction* $\wedge$ are the standard Boolean connectives and $\mathbf{U}_{[a,b]}$ is the *until* operator, with $a, b \in \mathbb{Q}$ and $a < b$. As customary, we can derive the *disjunction* operator $\vee$ by De Morgan's law and the *eventually* (a.k.a. future) operator $\mathbf{F}_{[t_1,t_2]}$ and the *always* (a.k.a. globally) operator $\mathbf{G}_{[t_1,t_2]}$ operators from the until operator.

Semantics. STL can be given not only the classic Boolean notion of *satisfaction*, denoted by $s(\varphi, \xi, t) = 1$ if ξ at time t satisfies φ, and 0 otherwise, but also a quantitative one, denoted by $\rho(\varphi, \xi, t)$. This measures the quantitative level of satisfaction of a formula for a given trajectory, evaluating how "robust" is the satisfaction of φ with respect to perturbations in the signal [15]. The quantitative semantics is defined recursively as follows:

$$\begin{aligned}
&\rho(\pi, \xi, t) && = f_\pi(\xi(t)) \qquad \text{for } \pi(x_1, ..., x_n) = \big(f_\pi(x_1, ..., x_n) \geq 0\big) \\
&\rho(\neg\, \varphi, \xi, t) && = -\,\rho(\varphi, \xi, t) \\
&\rho(\varphi_1 \,\wedge\, \varphi_2, \xi, t) && = \min\big(\rho(\varphi_1, \xi, t), \rho(\varphi_2, \xi, t)\big) \\
&\rho(\varphi_1 \mathbf{U}_{[a,b]} \varphi_2, \xi, t) && = \max_{t' \in [a+t, b+t]} \Big(\min\big(\rho(\varphi_2, \xi, t'), \min_{t'' \in [t,t']} \rho(\varphi_1, \xi, t'')\big)\Big)
\end{aligned}$$

Soundness and Completeness Robustness is compatible with satisfaction in that it complies with the following soundness property: if $\rho(\varphi, \xi, t) > 0$ then $s(\varphi, \xi, t) = 1$; and if $\rho(\varphi, \xi, t) < 0$ then $s(\varphi, \xi, t) = 0$. If the robustness is 0, both

[5] The whole framework can be easily relaxed to piecewise continuous càdlàg trajectories endowed with the Skorokhod topology and metric [8].

[6] Results are easily generalizable to predicates defined by piecewise continuous càdlàg functions.

satisfaction and the opposite may happen, but either way only non-robustly: there are arbitrarily small perturbations of the signal so that the satisfaction changes[7]. In fact, it complies also with a completeness property that ρ measures how robust the satisfaction of a trajectory is with respect to perturbations, see [15] for more detail.

Stochastic process in this context is a probability space $\mathcal{M} = (\mathcal{T}, \mathcal{A}, \mu)$, where $\mathcal{T}$ is a trajectory space and μ is a probability measure on a σ-algebra $\mathcal{A}$ over $\mathcal{T}$. Note that the definition is essentially equivalent to the standard definition of a stochastic process as a collection $\{D_t\}_{t \in I}$ of random variables, where $D_t(\xi) \in D$ is the signal $\xi(t)$ at time t on ξ [8]. The only difference is that we require, for simplicity[8], the signal be continuous.

Expected robustness and satisfaction probability. Given a stochastic process $\mathcal{M} = (\mathcal{T}, \mathcal{A}, \mu)$, we define the *expected robustness* $R_{\mathcal{M}} : \mathcal{P} \times I \to \mathbb{R}$ as

$$R_{\mathcal{M}}(\varphi, t) := \mathbb{E}_{\mathcal{M}}[\rho(\varphi, \xi, t)] = \int_{\xi \in \mathcal{T}} \rho(\varphi, \xi, t) d\mu(\xi) \, .$$

The qualitative counterpart of the expected robustness is the *satisfaction probability* $S(\varphi)$, i.e. the probability that a trajectory generated by the stochastic process $\mathcal{M}$ satisfies the formula φ: $S_{\mathcal{M}}(\varphi, t) := \mathbb{E}_{\mathcal{M}}[s(\varphi, \xi, t)] = \int_{\xi \in \mathcal{T}} s(\varphi, \xi, t) d\mu(\xi)$.[9] Finally, when $t = 0$ we often drop the parameter t from all these functions.

2.2 Kernel Crash Course

We recall the needed background for readers less familiar with machine learning.

Learning linear models. Linear predictors take the form of a vector of weights, intuitively giving positive and negative importance to features. A predictor given by a vector $\boldsymbol{w} = (w_1, \ldots, w_d)$ evaluates a data point $\boldsymbol{x} = (x_1, \ldots, x_d)$ to $w_1 x_1 + \cdots + w_d x_d = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$. To use it as a classifier, we can, for instance, take the sign of the result and output yes iff it is positive; to use it as a regressor, we can simply output the value. During learning, we are trying to separate, respectively approximate, the training data $\boldsymbol{x_1}, \ldots \boldsymbol{x_N}$ with a linear predictor, which corresponds to solving an optimization problem of the form (f is a suitable loss)

$$\underset{\boldsymbol{w} \in \mathbb{R}^d}{\arg\min} \, f(\langle \boldsymbol{w}, \boldsymbol{x_1} \rangle, \ldots, \langle \boldsymbol{w}, \boldsymbol{x_N} \rangle, \langle \boldsymbol{w}, \boldsymbol{w} \rangle)$$

where the possible, additional last term comes from regularization (preference of simpler weights, with lots of zeros in $\boldsymbol{w}$).

[7] The satisfaction of subformulae changes and, provided the predicates are "independent" of each other, the satisfaction of the whole formula, too.

[8] Again, this assumption can be relaxed since continuous functions are dense in the Skorokhod space of càdlàg functions.

[9] As argued above, this is essentially equivalent to integrating the indicator function of robustness being positive since a formula has robustness exactly zero only with probability zero as we sample all values from continuous distributions.

Need for a feature map $\varPhi$: *Input* $\to \mathbb{R}^n$. In order to learn, the input object first needs to be transformed to a vector of numbers. For instance, consider learning the logical exclusive-or function (summation in $\mathbb{Z}_2$) $y = x_1 \oplus x_2$. Seeing true as 1 and false as 0 already transforms the input into elements of $\mathbb{R}^2$. However, observe that there is no linear function separating sets of points $\{(0,0),(1,1)\}$ (where xor returns true) and $\{(0,1),(1,0)\}$ (where xor returns false). In order to facilitate learning by linear classifiers, richer feature space may be needed than what comes directly with the data. In our example, we can design a feature map to a higher-dimensional space using $\varPhi : (x_1, x_2) \mapsto (x_1, x_2, x_1 \cdot x_2)$. Then e.g. $x_3 \leq \frac{x_1+x_2-1}{2}$ holds in the new space iff $x_1 \oplus x_2$ and we can learn this linear classifier.

Another example can be seen in Fig. 1. The inner circle around zero cannot be linearly separated from the outer ring. However, considering $x_3 := x_1^2 + x_2^2$ as an additional feature turns them into easily separable lower and higher parts of a paraboloid.

In both examples, a feature map $\varPhi$ mapping the input to a space with higher dimension ($\mathbb{R}^3$), was used. Nevertheless, two issues arise:

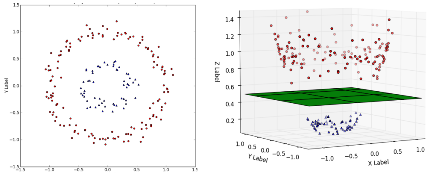


Fig. 1. An example illustrating the need for feature maps in linear classification [20].

1. What should be the features? Where do we get good candidates?
2. How to make learning efficient if there are too many features?

On the one hand, identifying the right features is hard, so we want to consider as many as possible. On the other hand, their number increases the dimension and thus decreases the efficiency both computationally and w.r.t. the number of samples required.

Kernel trick. Fortunately, there is a way to consider a huge amount of features, but with efficiency independent of their number (and dependent only on the amount of training data)! This is called the kernel trick. It relies on two properties of linear classifiers:

- The optimization problem above, after the feature map is applied, takes the form
$$\arg\min_{\boldsymbol{w} \in \mathbb{R}^n} f\big(\langle \boldsymbol{w}, \varPhi(\boldsymbol{x_1})\rangle, \ldots, \langle \boldsymbol{w}, \varPhi(\boldsymbol{x_N})\rangle, \langle \boldsymbol{w}, \boldsymbol{w}\rangle\big)$$
- Representer theorem: The optimum of the above can be written in the form
$$\boldsymbol{w}^* = \sum_{i=1}^{N} \alpha_i \varPhi(\boldsymbol{x_i})$$
Intuitively, anything orthogonal to training data cannot improve precision of the classification on the training data, and only increases $||\boldsymbol{w}||$, which we try to minimize (regularization).

Consequently, plugging the latter form into the former optimization problem yields an optimization problem of the form (g is a suitable loss derived from f):

$$\operatorname*{arg\,min}_{\boldsymbol{\alpha}\in\mathbb{R}^N} g\big(\boldsymbol{\alpha}, \langle \Phi(\boldsymbol{x_i}), \Phi(\boldsymbol{x_j})\rangle_{1\leq i,j\leq N}\big)$$

In other words, optimizing weights $\boldsymbol{\alpha}$ of expressions where data only appear in the form $\langle \Phi(\boldsymbol{x}_i), \Phi(\boldsymbol{x}_j)\rangle$. Therefore, we can take all features in $\Phi(\boldsymbol{x}_i)$ into account *if*, at the same time, we can efficiently evaluate the kernel function

$$k : (\boldsymbol{x}, \boldsymbol{y}) \mapsto \langle \Phi(\boldsymbol{x}), \Phi(\boldsymbol{y})\rangle$$

i.e. *without* explicitly constructing $\Phi(\boldsymbol{x})$ and $\Phi(\boldsymbol{y})$. Then we can efficiently learn the predictor on the rich set of features. Finally, when the predictor is applied to a new point $\boldsymbol{x}$, we only need to evaluate the expression

$$\langle \boldsymbol{w}, \Phi(\boldsymbol{x})\rangle = \sum_{i=1}^{N} \alpha_i \langle \Phi(\boldsymbol{x_i}), \Phi(\boldsymbol{x})\rangle = \sum_{i=1}^{N} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x})$$

3 Overview of Our Approach and Results

In this section, we describe what our tasks are if we want to apply the kernel trick in the setting of temporal formulae, what our solution ideas are, and where in the paper they are fully worked out.

1. *Design the kernel function:* define a similarity measure for STL formulae and prove it takes the form $\langle \Phi(\cdot), \Phi(\cdot)\rangle$
   (a) *Design an embedding of formulae into a Hilbert space* (vector space with possibly infinite dimension) ([10], Thm.3 in App.B proves this is well defined): Although learning can be applied also to data with complex structures such as graphs, the underlying techniques typically work on vectors. How do we turn a formula into a vector?
   Instead of looking at the syntax of the formula, we can look at its semantics. Similarly to Boolean satisfaction, where a formula can be identified with its language, i.e., the set $\mathcal{T} \to 2 \cong 2^{\mathcal{T}}$ of trajectories that satisfy it, we can regard an STL formula φ as a map $\rho(\varphi, \cdot) : \mathcal{T} \to \mathbb{R} \cong \mathbb{R}^{\mathcal{T}}$ of trajectories to their robustness. Observe that this is a real function, i.e., an *infinite-dimensional* vector of reals. Although explicit computations with such objects are problematic, kernels circumvent the issue. In summary, we have the implicit features given by the map:

   $$\varphi \overset{\Phi}{\mapsto} \rho(\varphi, \cdot)$$

   (b) *Design similarity on the feature representation (in Sec. 4.1):* Vectors' similarity is typically captured by their scalar product $\langle \boldsymbol{x}, \boldsymbol{y}\rangle = \sum_i x_i y_i$ since it gets larger whenever the two vectors "agree" on a component. In complete analogy, we can define for infinite-dimensional vectors (i.e.

functions) f, g their "scalar product" $\langle f, g\rangle = \int f(x)g(x)\,dx$. Hence we want the kernel to be defined as

$$k(\varphi, \psi) = \langle \rho(\varphi, \cdot), \rho(\psi, \cdot)\rangle = \int_{\xi\in\mathcal{T}} \rho(\varphi, \xi)\rho(\psi, \xi)\, d\xi$$

(c) *Design a measure on trajectories (Sec. 4.2):* Compared to finite-dimensional vectors, where in the scalar product each component is taken with equal weight, integrating over uncountably many trajectories requires us to put a finite measure on them, according to which we integrate. Since, as a side effect, it necessarily expresses their importance, we define a probability measure μ_0 preferring "simple" trajectories, where the signals do not change too dramatically (the so-called total variation is low). This finally yields the definition of the kernel as[10]

$$\boxed{k(\varphi, \psi) = \int_{\xi\in\mathcal{T}} \rho(\varphi, \xi)\rho(\psi, \xi)\, d\mu_0(\xi) \qquad\qquad (1)}$$

2. *Learn the kernel (Sec. 5.1):*
   (a) *Get training data $\boldsymbol{x_i}$:* The formulae for training should be chosen according to the same distribution as they are coming in the final task of prediction. Since that distribution is unknown, we assume at least a general preference of simple formulae and thus design a probability distribution $\mathcal{F}_0$, preferring formulae with simple syntax trees (see Section 5.1). We also show that several hundred formulae are sufficient for practically precise predictions.
   (b) *Compute the "correlation" of the data $\langle\phi(\boldsymbol{x_i}), \phi(\boldsymbol{x_j})\rangle$ by kernel $k(\boldsymbol{x_i}, \boldsymbol{x_j})$:* Now we evaluate (1) for all the data pairs. Since this involves an integral over all trajectories, we simply approximate it by Monte Carlo: We choose a number of trajectories according to μ_0 and sum the values for those. In our case, 10 000 provide a very precise approximation.
   (c) *Optimize the weights $\boldsymbol{\alpha}$ (using values from (b)):* Thus we get the most precise linear classifier given the data, but penalizing too "complicated" ones since they tend to overfit and not generalize well (so-called regularization). Recall that the dimension of $\boldsymbol{\alpha}$ is the size of the training data set, not the infinity of the Hilbert space.
3. *Evaluate the predictive power of the kernel* and thus implicitly the kernel function design:
   – We evaluate the accuracy of predictions of robustness for single trajectories (Sec. 5.2), the expected robustness on a stochastic system and the corresponding Boolean notion of satisfaction probability (Sec. 5.3). Moreover, we show that there is no need to derive kernel for each stochastic process separately depending on their probability spaces, but the one

[10] On the conceptual level; technically, additional normalization and Gaussian transformation are performed to ensure usual desirable properties, see Cor. 1 in Sec. 4.1.

derived from the generic μ_0 is sufficient and, surprisingly, even more accurate (Sec. 5.4).
- Besides the experimental evaluation, we provide a PAC bound on our methods in terms of Rademacher complexity [26] (Sec. 4.4).

4 A Kernel for Signal Temporal Logic

In this section, we sketch the technical details of the construction of the STL kernel, of the correctness proof, and of PAC learning bounds. More details on the definition, including proofs, are provided in [10], Appendix B.

4.1 Definition of STL Kernel

Let us fix a formula $\varphi \in \mathcal{P}$ in the STL formulae space and consider the robustness $\rho(\varphi, \cdot, \cdot) : \mathcal{T} \times I \to \mathbb{R}$, seen as a real-valued function on the domain $\mathcal{T} \times I$, where $I \subset \mathbb{R}$ is a bounded interval, and $\mathcal{T}$ is the trajectory space of continuous functions. The STL kernel is defined as follows.

Definition 1. *Fixing a probability measure μ_0 on $\mathcal{T}$, we define the STL-kernel*

$$k'(\varphi, \psi) = \int_{\xi \in \mathcal{T}} \int_{t \in I} \rho(\varphi, \xi, t)\rho(\psi, \xi, t) dt d\mu_0$$

The integral is well defined as it corresponds to a scalar product in a suitable Hilbert space of functions. Formally proving this, and leveraging foundational results on kernel functions [26], in [10], Appendix B, we prove the following:

Theorem 1. *The function k' is a proper kernel function.*

In the previous definition, we can fix time to $t = 0$ and remove the integration w.r.t. time. This simplified version of the kernel is called *untimed*, to distinguish it from the *timed* one introduced above.

In the rest of the paper, we mostly work with two derived kernels, k_0 and k:

$$k_0(\varphi, \psi) = \frac{k'(\varphi, \psi)}{\sqrt{k'(\varphi, \varphi) k'(\psi, \psi)}} \qquad k(x, y) = \exp\left(-\frac{1 - 2k_0(x, y)}{\sigma^2}\right). \tag{2}$$

The normalized kernel k_0 rescales k' to guarantee that $k(\varphi, \varphi) \geq k(\varphi, \psi)$, $\forall \varphi, \psi \in \mathcal{P}$. The Gaussian kernel k, additionally, allows us to introduce a soft threshold σ^2 to fine tune the identification of significant similar formulae in order to improve learning. The following proposition is straightforward in virtue of the closure properties of kernel functions [26]:

Corollary 1. *The functions k_0 and k are proper kernel functions.*

4.2 The Base Measure μ_0

In order to make our kernel meaningful and not too expensive to compute, we endow the trajectory space $\mathcal{T}$ with a probability distribution such that more complex trajectories are less probable. We use the total variation [29] of a trajectory[11] and the number of changes in its monotonicity as indicators of its "complexity".

Because later we use the probability measure μ_0 for Monte Carlo approximation of the kernel k, it is advantageous to define μ_0 algorithmically, by providing a *sampling algorithm*. The algorithm samples from continuous piece-wise linear functions, a dense subset of $\mathcal{T}$, and is described in detail in [10], Appendix A. Essentially, we simulate the value of a trajectory at discrete steps Δ, for a total of N steps (equal to 100 in the experiments) by first sampling its total variation distance from a squared Gaussian distribution, and then splitting such total variation into the single steps, changing sign of the derivative at each step with small probability q. We then interpolate linearly between consecutive points of the discretization and make the trajectory continuous piece-wise linear.

In Section 5.4, we show that using this simple measure still allows us to make predictions with remarkable accuracy even for other stochastic processes on $\mathcal{T}$.

4.3 Normalized Robustness

Consider the predicates $x_1 - 10 \geq 0$ and $x_1 - 10^7 \geq 0$. Given that we train and evaluate on μ_0, whose trajectories typically take values in the interval $[-3, 3]$ (see also [10], Appendix A), both predicates are essentially equivalent for satisfiability. However, their robustness on the same trajectory differs by orders of magnitude. This very same effect, on a smaller scale, happens also when comparing $x_1 \geq 10$ with $x_1 \geq 20$. In order to ameliorate this issue and make the learning less sensitive to outliers, we also consider a *normalized robustness*, where we rescale the value of the secondary (output) signal to $(-1, 1)$ using a sigmoid function. More precisely, given an atomic predicate $\pi(x_1, ..., x_n) = (f_\pi(x_1, ..., x_n) \geq 0)$, we define $\hat{\rho}(\pi, \xi, t) = \tanh\left(f_\pi(x_1, ..., x_n)\right)$. The other operators of the logic follow the same rules of the standard robustness described in Section 2.1. Consequently, both $x_1 - 10 \geq 0$ and $x_1 - 10^7 \geq 0$ are mapped to very similar robustness for typical trajectories w.r.t. μ_0, thus reducing the impact of outliers.

4.4 PAC Bounds for the STL Kernel

Probably Approximately Correct (PAC) bounds [26] for learning provide a bound on the generalization error on unseen data (known as risk) in terms of the training loss plus additional terms which shrink to zero as the number of samples grows. These additional terms typically depend also on some measure of the complexity of the class of models we consider for learning (the so-called hypothesis space),

[11] The total variation of function f defined on $[a, b]$ is $V_a^b(f) = \sup_{P \in \mathbf{P}} \sum_{i=0}^{n_P - 1} |f(x_{i+1}) - f(x_i)|$, where $\mathbf{P} = \{P = \{x_0, \ldots, x_{n_P}\} \mid P \text{ is a partition of } [a, b]\}$.

which ought to be finite. The bound holds with probability $1-\delta$, where $\delta > 0$ can be set arbitrarily small at the price of the bound getting looser.

In the following, we will state a PAC bound for learning with STL kernels for classification. A bound for regression, and more details on the classification bound, can be found in [10], Appendix C. We first recall the definition of the risk L and the empirical risk $\hat{L}$ for classification. The former is an average of the zero-one loss over the data generating distribution p_{data}, while the latter averages over a finite sample D of size m of p_{data}. Formally,

$$L(h) = \mathbb{E}_{\varphi \sim p_{data}} \left[\mathbb{I}\big(h(\varphi) \neq y(\varphi)\big) \right] \quad \text{and} \quad \hat{L}_D(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}\big(h(\varphi_i) \neq y(\varphi_i)\big),$$

where $y(\varphi)$ is the actual class (truth value) associated with φ, in contrast to the predicted class $h(\varphi)$, and $\mathbb{I}$ is the indicator function.

The major issue with PAC bounds for kernels is that we need to constrain in some way the model complexity. This is achieved by requesting the functions that can be learned have a bounded norm. We recall that the norm $\|h\|_{\mathbb{H}}$ of a function h obtainable by kernel methods, i.e. $h(\varphi) = \sum_{i=1}^{N} \alpha_i k(\varphi_i, \varphi)$, is $\|h\|_{\mathbb{H}} = \boldsymbol{\alpha}^T K \boldsymbol{\alpha}$, where K is the Gram matrix (kernel evaluated between all pairs of input points, $K_{ij} = k(\varphi_i, \varphi_j)$). The following theorem, stating the bounds, can be proved by combining bounds on the Rademacher complexity for kernels with Rademacher complexity based PAC bounds, as we show in [10], Appendix C.

Theorem 2 (PAC bounds for Kernel Learning in Formula Space). *Let k be a kernel (e.g. normalized, exponential) for STL formulae $\mathcal{P}$, and fix $\Lambda > 0$. Let $y : \mathcal{P} \to \{-1, 1\}$ be a target function to learn as a classification task. Then for any $\delta > 0$ and hypothesis function h with $\|h\|_{\mathbb{H}} \leq \Lambda$, with probability at least $1-\delta$ it holds that*

$$L(h) \leq \hat{L}_D(h) + \frac{\Lambda}{\sqrt{m}} + 3\sqrt{\frac{\log \frac{2}{\delta}}{2m}}. \tag{3}$$

The previous theorem gives us a way to control the learning error, provided we restrict the full hypothesis space. Choosing a value of Λ equal to 40 (the typical value we found in experiments) and confidence 95%, the bound predicts around 650 000 samples to obtain an accuracy bounded by the accuracy on the training set plus 0.05. This theoretical a-priori bound is much larger than the training set sizes in the order of hundreds, for which we observe good performance in practice.

5 Experiments

We test the performance of the STL kernel in predicting (a) robustness and satisfaction on single trajectories, and (b) expected robustness and satisfaction probability estimated statistically from K trajectories. Besides, we test the kernel on trajectories sampled according to the a-priori base measure μ_0 and according to the respective stochastic models to check the generalization power of the generic μ_0-based kernel. Here we report the main results; for additional details as well as plots and tables for further ways of measuring the error, we refer the interested reader to [10], Appendix D.

Computation of the STL robustness and of the kernel was implemented in Python exploiting PyTorch [30] for parallel computation on GPUs. All the experiments were run on a AMD Ryzen 5000 with 16 GB of RAM and on a consumer NVidia GTX 1660Ti with 6 GB of DDR6 RAM. We run each experiment 1000 times for single trajectories and 500 for expected robustness and satisfaction probability where we use 5000 trajectories for each run. Where not indicated differently, each result is the mean over all experiments. Computational time is fast: the whole process of sampling from μ_0, computing the kernel, doing regression for training, test set of size 1000 and validation set of size 200, takes about 10 seconds on GPU. We use the following acronyms: RE = relative error, AE= absolute error, MRE = mean relative error, MAE = mean absolute error, MSE = mean square error.

5.1 Setting

To compute the kernel itself, we sampled 10 000 trajectories from μ_0, using the sampling method described in Section 4.2. As regression algorithm (for optimizing $\boldsymbol{\alpha}$ of Sections 2.2 and 3) we use the *Kernel Ridge Regression* (KRR) [27]. KRR was as good as, or superior, to other regression techniques (a comparison can be found in [10], Appendix D.1).

Training and test set are composed of M formulae sampled randomly according to the measure $\mathcal{F}_0$ given by a syntax-tree random recursive growing scheme (reported in detail in [10], Appendix D.1), where the root is always an operator node and each node is an atomic predicate with probability p_{leaf} (fixed in this experiments to 0.5), or, otherwise, another operator node (sampling the type using a uniform distribution). In these experiments, we fixed $M = 1000$.

Hyperparameters We vary several hyperparameters, testing their impact on errors and accuracy. Here we briefly summarize the results.
- The impact of *formula complexity*: We vary the parameter p_{leaf} in the formula generating algorithm in the range $[0.2, 0.3, 0.4, 0.5]$ (average formula size around $[100, 25, 10, 6]$ nodes in the syntax tree), but only a slight increase in the median relative error is observed for more complex formulae: $[0.045, 0.037, 0.031, 0.028]$.
- The addition of *time bounds* in the formulae has essentially no impact on the performance in terms of errors.
- There is a very small improvement ($< 10\%$) using *integrating signals w.r.t. time* (timed kernel) vs using only robustness at time zero (untimed kernel), but at the cost of a 5-fold increase in computational training time.

- *Size of training set*: The error in estimating robustness decreases as we increase the amount of training formulae, see Fig. 2. However, already for a few hundred formulae, the predictions are quite accurate.

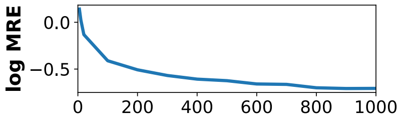


Fig. 2. MRE of predicted average robustness vs the size of the training set.

- *Exponential kernel k* gives a 3-fold improvement in accuracy w.r.t. normalized kernel k_0.
- *Dimensionality of signals*: Error tends to increase linearly with dimensionality. For 1000 formulae in the training set, from dimension 1 to 5, MRE is [0.187, 0.248, 0.359, 0.396, 0.488] and MAE is [0.0537, 0.0735, 0.0886, 0.098, 0.112].

5.2 Robustness and Satisfaction on Single Trajectories

In this experiment, we predict the Boolean satisfiability of a formula using as a discriminator the sign of the robustness. We generate the training and test set of formulae using $\mathcal{F}_0$, and the function sampling trajectories from μ_0 with dimension $n = 1, 2, 3$, using an independent sample than the one for evaluating the kernel. We evaluate the standard robustness ρ and the normalized one $\hat{\rho}$ of each trajectory for each formula in the training and test sets. We then predict ρ and $\hat{\rho}$ for the test set and check if the sign of the predicted robustness agrees with that of the true one, which is a proxy for satisfiability, as discussed previously. Accuracy and distribution of the $\log_{10}$ MRE over all experiments are reported in Fig. 3. Results are good for both but the normalized robustness performs always better. Accuracy is always greater than 0.96 and gets slightly worse when increasing the dimension. We report the mean of quantiles of ρ and $\hat{\rho}$ for RE and AE for n=3 (the toughest case) in Table 1 (top two rows). Errors for the normalized one are also always lower and slightly worsen when increasing the dimension.

In Fig. 4 (left), we plot the true standard robustness for random test formulae in contrast to their predicted values and the corresponding log RE. Here we

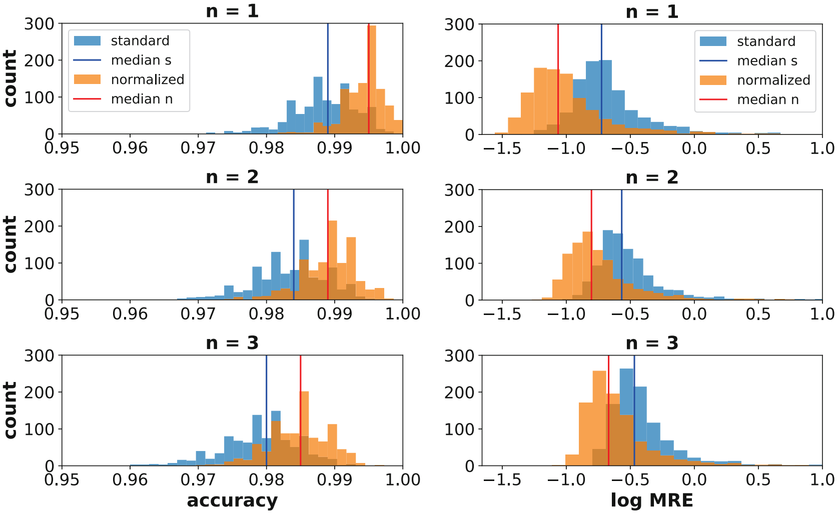


Fig. 3. Accuracy of satisfiability prediction (left) and log_{10} of the MRE (right) over all 1000 experiments for standard and normalized robustness for samples from μ_0 with dimensionality of signals $n = 1, 2, 3$. (Note the logarithmic scale, with log value of -1 corresponding to 0.1 of the standard non-logarithmic scale.)

Table 1. Mean of quantiles for RE and AE over all experiments for prediction of the standard and normalized robustness (ρ, $\hat{\rho}$), expected robustness (R, $\hat{R}$), the satisfaction probability (S) with trajectories sampled from μ_0 and signals with dimensionality n=3, and of the normalized expected robustness on trajectories sampled from *Immigration* (1 dim), *Isomerization* (2 dim), and *Transcription* (3 dim)

	relative error (RE)						absolute error (AE)			
	5perc	1quart	median	3quart	95perc	99perc	1quart	median	3quart	99perc
ρ	0.0035	0.018	0.045	0.141	0.870	4.28	0.016	0.039	0.105	0.689
$\hat{\rho}$	0.0008	0.001	0.006	0.019	0.564	2.86	0.004	0.012	0.039	0.286
R	0.0045	0.021	0.044	0.103	0.548	2.41	0.013	0.029	0.070	0.527
$\hat{R}$	0.0006	0.003	0.007	0.020	0.133	0.55	0.001	0.003	0.007	0.065
S	0.0005	0.003	0.008	0.030	0.586	81.8	0.001	0.003	0.007	0.072
$\hat{R}$ imm	0.0053	0.0067	0.016	0.049	0.360	1.83	0.0037	0.008	0.019	0.151
$\hat{R}$ iso	0.0030	0.0092	0.026	0.091	0.569	2.74	0.0081	0.021	0.057	0.460
$\hat{R}$ trancr	0.0072	0.0229	0.071	0.240	1.490	7.55	0.018	0.049	0.12	0.680

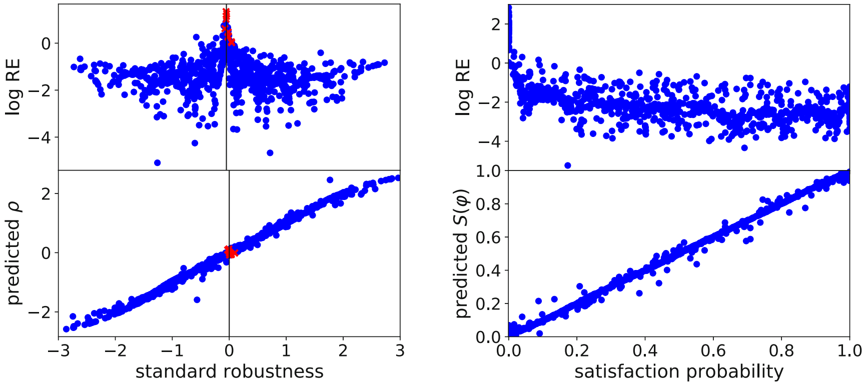


Fig. 4. (left) True standard robustness vs predicted values and RE on single trajectories sampled from μ_0. The misclassified formulae are the red crosses. (right) Satisfaction probability vs predicted values and RE (again for a single experiment).

can clearly observe that the misclassified formulae (red crosses) tend to have a robustness close to zero, where even tiny absolute errors unavoidably produce large relative errors and frequent misclassification.

We test our method also on three specifications of the ARCH-COMP 2020 [16], to show that it works well even on real formulae. We obtain still good results, with an accuracy equal to 1, median AE = 0.0229, and median RE = 0.0316 in the worst case (the AT1 of the Automatic Transmission (AT) Benchmark, see [10], Appendix D.2).

5.3 Expected Robustness and Satisfaction Probability

In these experiments, we approximate the expected standard $R(\varphi)$ and normalized $\hat{R}(\varphi)$ and the satisfaction probability $S(\phi)$ using a fixed set of 5000 trajectories sampled according to μ_0, independent of the one used to compute the kernel, evaluating it for each formula in the training and test sets, and predicting $R(\varphi)$, $\hat{R}(\varphi)$ and $S(\phi)$ for the test set.

For the robustness, the mean of quantiles of RE and AE shows good results as can be seen in Table 1, rows 3–4. Values of MSE, MAE and MRE are smaller than those achieved on single trajectories with medians for n=3 equal to 0.0015, 0.064, and 0.2 for $R(\varphi)$ and 0.00021, 0.0067, and 0.048 for the $\hat{R}(\varphi)$. Normalized robustness continues to outperform the standard one.

For the satisfaction probability, values of MSE and MAE errors are very low, with a median for n=3 equal to 0.000247 for MSE and 0.0759 for MAE. MRE instead is higher and equal to 3.21. The reason can be seen in Fig. 4 (right), where we plot the satisfaction probability vs the relative error for a random experiment. We can see that all large relative errors are concentrated on formulae with satisfaction probability close to zero, for which even a small absolute deviation can cause large errors. Indeed the 95th percentile of RE is still pretty low, namely 0.586 (cf. Table 1, row 5), while we observe the 99th percentile of RE blowing up to 81.8 (at points of near zero true probability). This heavy tailed behaviour suggests to rely on median for a proper descriptor of typical errors, which is 0.008 (hence the typical relative error is less than 1%).

5.4 Kernel Regression on Other Stochastic Processes

The last aspect that we investigate is whether the definition of our kernel w.r.t. the fixed measure μ_0 can be used for making predictions also for other stochastic processes, i.e. without redefining and recomputing the kernel every time that we change the distribution of interest on the trajectory space.

Standardization. To use the same kernel of μ_0 we need to standardize the trajectories so that they have the same scale as our base measure. Standardization, by subtracting to each variable its sample mean and dividing by its sample standard deviation, will result in a similar range of values as that of trajectories sampled from μ_0, thus removing distortions due to the presence of different scales and allowing us to reason on the trajectories using thresholds like those generated by the STL sampling algorithm.

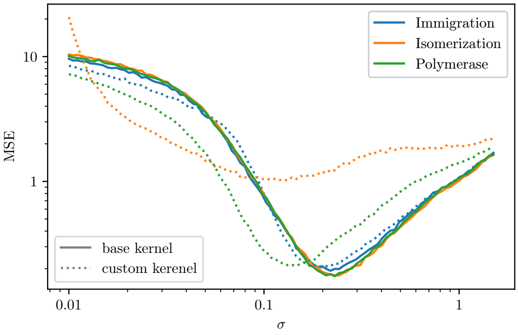


Fig. 5. Expected robustness prediction using the kernel evaluated according to the *base kernel*, and a *custom kernel*. We depict MSE as a function of the bandwidth σ of the Gaussian kernel (with both axes in logarithmic scale).

Performance of base and custom kernel. We consider three different stochastic models: *Immigration* (1 dim), *Isomerization* (2 dim) and *Polymerise* (2 dim), simulated using the Python library StochPy [22] (see also [10], Appendix D.5).

We compare the performance using the kernel evaluated according to the base measure μ_0 (base kernel), and a custom kernel computed replacing μ_0 with the measure on trajectories given by the stochastic model itself. Results show that the base kernel is still the best performing one, see Fig. 5. This can be explained by the fact that the measure μ_0 is broad in terms of coverage of the trajectory space, so even if two formulae are very similar, there will be, with a high probability, a set of trajectories for which the robustnesses of the two formulae are very different. This allows us to better distinguish among STL formulae, compared to models that tend to focus the probability mass on narrower regions of $\mathcal{T}$ as, for example, the Isomerization model, which is the model with the most homogeneous trajectory space and has indeed the worst performance.

Expected Robustness Setting is the same as for the corresponding experiment on μ_0. Instead of the Polymerase model, we consider here a *Transcription* model [22] (see also [10], Appendix D.5), to have also a 3-dimensional model. Results of quantile for RE and AE for the normalized robustness are reported in Table 1, bottom three rows. The results on the different models are remarkably promising, with the Transcription model (median RE 7%) performing a bit worse than Immigration and Isomerization (1.6% and 2.6% median RE). Similar experiments have been done also on single trajectories, where we obtain similar results as for the Expected Robustness [10], Appendix D.5.

6 Conclusions

To enable any learning over formulae, their features must be defined. We circumvented the typically manual and dubious process by adopting a more canonic, infinite-dimensional feature space, relying on the quantitative semantics of STL. To effectively work with such a space, we defined a kernel for STL. To further overcome artefacts of the quantitative semantics, we proposed several normalizations of the kernel. Interestingly, we can use *exactly* the same kernel with a fixed base measure over trajectories across different stochastic models, not requiring any access to the model. We evaluated the approach on realistic biological models from the stochpy library as well as on realistic formulae from Arch-Comp and concluded a good accuracy already with a few hundred training formulae.

Yet smaller training sets are possible through a wiser choice of the training formulae: one can incrementally pick formulae significantly different (now that we have a similarity measure on formulae) from those already added. Such active learning results in a better coverage of the formula space, allowing for a more parsimonious training set. Besides estimating robustness of concrete formulae, one can lift the technique to computing STL-based distances between stochastic models, given by differences of robustness over *all* formulae, similarly to [14]. To this end, it suffices to resort to a dual kernel construction, and build non-linear embeddings of formulae into finite-dimensional real spaces using the kernel-PCA techniques [27]. Our STL kernel, however, can be used for many other tasks, some of which we sketched in Introduction. Finally, to further improve its properties, another direction for future work is to refine the quantitative semantics so that equivalent formulae have the same robustness, e.g. using ideas like in [23].

References

1. Amortila, P., Bellemare, M.G., Panangaden, P., Precup, D.: Temporally extended metrics for markov decision processes. In: SafeAI@AAAI. CEUR Workshop Proceedings, vol. 2301. CEUR-WS.org (2019)
2. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: A complete quantitative deduction system for the bisimilarity distance on markov chains. Log. Methods Comput. Sci. **14**(4) (2018)
3. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R., Tang, Q., van Breugel, F.: Computing probabilistic bisimilarity distances for probabilistic automata. In: CONCUR. LIPIcs, vol. 140, pp. 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
4. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
5. Bartocci, E., Bortolussi, L., Nenzi, L., Sanguinetti, G.: System design of stochastic models using robustness of temporal properties. Theor. Comput. Sci. **587**, 3–25 (2015). https://doi.org/10.1016/j.tcs.2015.02.046, https://doi.org/10.1016/j.tcs.2015.02.046
6. Bartocci, E., Bortolussi, L., Sanguinetti, G.: Data-driven statistical learning of temporal logic properties. In: Proc. of FORMATS. pp. 23–37 (2014)
7. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Lectures on Runtime Verification, pp. 135–175. Springer (2018)
8. Billingsley, P.: Probability and measure. John Wiley & Sons (2008)
9. Bombara, G., Vasile, C.I., Penedo, F., Yasuoka, H., Belta, C.: A Decision Tree Approach to Data Classification using Signal Temporal Logic. In: Hybrid Systems: Computation and Control. pp. 1–10. ACM Press (2016). https://doi.org/10.1145/2883817.2883843
10. Bortolussi, L., Gallo, G.M., Křetínský, J., Nenzi, L.: Learning model checking and the kernel trick for signal temporal logic on stochastic processes. Tech. Rep. 2201.09928, arXiv (2022), https://arxiv.org/abs/2201.09928
11. Brezis, H.: Functional analysis, Sobolev spaces and partial differential equations. Springer Science & Business Media (2010)
12. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
13. Comaniciu, D., Meer, P.: Mean shift: A robust approach toward feature space analysis. IEEE Transactions on Pattern Analysis & Machine Intelligence **24**(5), 603–619 (2002)
14. Daca, P., Henzinger, T.A., Kretínský, J., Petrov, T.: Linear distances between markov chains. In: Desharnais, J., Jagadeesan, R. (eds.) CONCUR. LIPIcs, vol. 59, pp. 20:1–20:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.20, https://doi.org/10.4230/LIPIcs.CONCUR.2016.20
15. Donzé, A., Ferrere, T., Maler, O.: Efficient robust monitoring for stl. In: International Conference on Computer Aided Verification. pp. 264–279. Springer (2013)
16. Ernst, G., Arcaini, P., Bennani, I., Donze, A., Fainekos, G., Frehse, G., Mathesen, L., Menghi, C., Pedrielli, G., Pouzet, M., Yaghoubi, S., Yamagata, Y., Zhang, Z.: Arch-comp 2020 category report: Falsification. In: Frehse, G., Althoff, M. (eds.) ARCH20. 7th International Workshop on Applied Verification of Continuous and

Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 140–152. EasyChair (2020). https://doi.org/10.29007/trr1, https://easychair.org/publications/paper/ps5t

17. Fainekos, G., Hoxha, B., Sankaranarayanan, S.: Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with s-taliro. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification (RV). Lecture Notes in Computer Science, vol. 11757, pp. 27–47. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_3, https://doi.org/10.1007/978-3-030-32079-9_3
18. Haghighi, I., Mehdipour, N., Bartocci, E., Belta, C.: Control from signal temporal logic specifications with smooth cumulative quantitative semantics. In: 58th IEEE Conference on Decision and Control, CDC 2019, Nice, France, December 11-13, 2019. pp. 4361–4366. IEEE (2019). https://doi.org/10.1109/CDC40024.2019.9029429, https://doi.org/10.1109/CDC40024.2019.9029429
19. Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P.J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., Walker, A.: The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR **abs/1904.07736** (2019)
20. Kim, E.: Everything you wanted to know about the kernel trick (but were too afraid to ask). https://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html, accessed on Jan 20, 2021
21. Kretínský, J., Manta, A., Meggendorfer, T.: Semantic labelling and learning for parity game solving in LTL synthesis. In: ATVA. Lecture Notes in Computer Science, vol. 11781, pp. 404–422. Springer (2019)
22. Maarleveld, T.R., Olivier, B.G., Bruggeman, F.J.: Stochpy: a comprehensive, user-friendly tool for simulating stochastic biological processes. PloS one **8**(11), e79345 (2013)
23. Madsen, C., Vaidyanathan, P., Sadraddini, S., Vasile, C.I., DeLateur, N.A., Weiss, R., Densmore, D., Belta, C.: Metrics for signal temporal logic formulae. In: 2018 IEEE Conference on Decision and Control (CDC). pp. 1542–1547. IEEE (2018)
24. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Proc. FORMATS (2004)
25. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)
26. Mohri, M., Rostamizadeh, A., Talwalkar, A.: Foundations of machine learning. The MIT Press, Cambridge, Massachusetts, second edition edn. (2018)
27. Murphy, K.P.: Machine learning: a probabilistic perspective. MIT press (2012)
28. Nenzi, L., Silvetti, S., Bartocci, E., Bortolussi, L.: A robust genetic algorithm for learning temporal specifications from data. In: McIver, A., Horváth, A. (eds.) QEST. Lecture Notes in Computer Science, vol. 11024, pp. 323–338. Springer (2018). https://doi.org/10.1007/978-3-319-99154-2_20, https://doi.org/10.1007/978-3-319-99154-2_20
29. Pallara, D Ambrosio, L., Fusco, N.: Functions of bounded variation and free discontinuity problems. Oxford University Press, Oxford (2000)
30. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. In: NIPS 2017 Workshop on Autodiff (2017), https://openreview.net/forum?id=BJJsrmfCZ
31. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning. MIT Press (2006)

32. Shawe-Taylor, J., Cristianini, N.: Kernel methods for pattern analysis. Cambridge Univ Pr (2004)
33. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. Lecture Notes in Computer Science, vol. 2404, pp. 223–235. Springer (2002)

Verification Inference

Inferring Interval-Valued Floating-Point Preconditions*

Jonas Krämer[1], Lionel Blatter[2], Eva Darulova[3] (✉), and Mattias Ulbrich[2]

[1] itemis AG, Stuttgart, Germany [§], `jonas.kraemer@itemis.com`
[2] KIT, Karlsruhe, Germany `{lionel.blatter,ulbrich}@kit.edu`
[3] Uppsala University, Uppsala, Sweden[†], `eva.darulova@it.uu.se`

Abstract. Aggregated roundoff errors caused by floating-point arithmetic can make numerical code highly unreliable. Verified postconditions for floating-point functions can guarantee the accuracy of their results under specific preconditions on the function inputs, but how to systematically find an adequate precondition for a desired error bound has not been explored so far. We present two novel techniques for automatically synthesizing preconditions for floating-point functions that guarantee that user-provided accuracy requirements are satisfied. Our evaluation on a standard benchmark set shows that our approaches are complementary and able to find accurate preconditions in reasonable time.

1 Introduction

Floating-point arithmetic as defined by the IEEE 754 standard [18] is widely used to approximate real arithmetic in embedded or scientific computing applications. While allowing highly efficient computations, the limited precision of floating-point numbers introduces roundoff errors in every single operation [24]. The aggregated errors in computations where such rounding happens repeatedly are challenging to understand and predict intuitively, so that a variety of techniques and tools [10,14,11,29,21,22] have been developed that bound worst-case roundoff errors. These techniques assume a given floating-point precision, e.g. uniform double precision and a precondition $\psi(\bar{x})$ that bounds a function's possibly multivariate parameters $(\bar{x})$, and automatically compute an upper-bound ε on the function result's absolute roundoff error $(f_{err}(\bar{x}))$[4]:

$$\forall \bar{x}.\ \ \psi(\bar{x}) \rightarrow f_{err}(\bar{x}) \leq \varepsilon \tag{1}$$

Answering the inverse question can be equally useful: given a desired roundoff error bound and precision, for which inputs will the computation's result be

[§] Part of this work was done while the author was at KIT, Germany.
[†] Part of this work was done while the author was at MPI-SWS, Germany.
[*] Part of this work was funded by the AESC project supported by the Ministry of Science, Research and Arts Baden-Württemberg (Ref: 33-7533.-9-10/20/1).
[4] We provide more formalization details in the next section.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 303–321, 2022.
https://doi.org/10.1007/978-3-030-99524-9_16

at least this accurate? That is, given a postcondition specifying the error bound for a floating-point function's result, we want to infer a suitable precondition ψ. Such preconditions can be useful for modular verification of larger floating-point programs, or for efficient implementations: for inputs that satisfy the generated precondition, the function can be evaluated using e.g. efficient double-precision floating-point arithmetic, instead of a more accurate but significantly more expensive arbitrary-precision arithmetic [2] that would have to be used for the remaining input space.

Outside the analysis of floating-point software, the automatic synthesis of preconditions for software components is not a new field of study. Dijkstra's weakest precondition calculus [12], while not originally intended to be used for specification inference, can generate weakest preconditions. However, when applied to a floating-point function, it creates a precondition that still contains the floating-point arithmetic of the analyzed program and is, thus, not simpler than the program itself. Recent approaches (targeting non-floating-point programs) for specification inference [23,28,7,13] similarly do not attempt to abstract from arithmetic operations and their inaccuracies.

This paper introduces two novel techniques for synthesizing *sound and abstract preconditions* for floating-point functions. The inferred preconditions $\psi(\bar{x})$ are sound, by which we mean that they are guaranteed to satisfy Eq. (1) for a user-specified error bound ε. The preconditions are abstract in the sense that they do not contain any floating-point arithmetic operations.

We choose to synthesize *interval-valued* preconditions that bound each function parameter by a lower and an upper bound, i.e. $x \in [a, b]$. Such preconditions avoid floating-point arithmetic, and thus roundoff errors, as evaluating them requires only comparisons with constants. Our preconditions are relatively simple on purpose to ensure compatibility with current sound roundoff verification techniques that internally rely on interval-based abstractions. While more complex, e.g. nonlinear, constraints may be more precise, they are not well-supported by state-of-the-art verifiers and thus their benefit would be (currently) lost.

While we aim to synthesize weak preconditions that cover much of the input space, weakest preconditions are not necessarily helpful in the context of floating-point computations. The reason is that the space of inputs satisfying a postcondition—especially one bounding the roundoff error—is in general highly discontinuous due to the discrete nature of floating-point arithmetic. A weakest precondition would thus consist of a large conjunction, with individual terms often covering only a few values, and would hence not be practically useful. Instead, we aim to find preconditions that balance precision (are as weak as possible) and complexity (are simple and can be evaluated efficiently).

We are not aware of an existing approach for generating such sound floating-point preconditions; we thus choose to introduce and explore two quite different techniques that build on existing dynamic and static floating-point analyses in a novel way. Both approaches start by dynamically *sampling* the analyzed function in order to find likely precondition candidates and then use a verification backend to refine them until their soundness can be guaranteed. The first *recur-*

sive subdivision approach does this by recursively subdividing the input space into increasingly smaller cells, discarding those where sampling shows that the postcondition is not satisfied for the contained inputs, and attempting to verify the rest. Since such generated preconditions may still contain a large number of discontinuous subdomains, we further present an optimization algorithm that soundly approximates the preconditions with significantly simpler expressions that can be evaluated more efficiently. The second *classification tree* approach learns areas of inputs for which the postcondition holds based on a classification tree learned from the dynamic samples, and iteratively *refines* verified preconditions in these areas.

Our approaches guarantee soundness of the generated preconditions by verifying each individual interval domain in the preconditions using a sound floating-point roundoff error analyzer. Our approach is generic in the choice of this tool; we integrate the floating-point verification framework Daisy [10].

We evaluate and compare our proposed approaches on benchmarks from the standard floating-point benchmark suite FPBench [8] and show that the approaches are able to find adequate preconditions that (1) are syntactically simple and cheap to evaluate and (2) are relatively weak, i.e. good approximations of the weakest preconditions covering large areas of the input space, thus balancing complexity and permissiveness. For most benchmarks, our approaches find preconditions in under 20 minutes (and often significantly faster). We demonstrate a possible application of our inferred preconditions for performance improvements on a case study using a kernel from a real-world material sciences code that inspired this work.

Contributions In summary, this paper makes the following contributions:

- Two independent novel inference algorithms that generate interval-valued preconditions for floating-point functions. They are the first of their kind.
- An open-source implementation of both approaches as part of the Daisy floating-point analysis framework.
- An extensive evaluation on 99 benchmarks and a case study showing the effectiveness of our precondition inference.

2 Overview

Before explaining our approaches in detail, we provide a high-level overview using an example. Consider the two-dimensional function `himmilbeau` from the floating-point benchmark suite FPBench [8], introduced to evaluate optimization algorithms [16], and defined as

$$\hat{f}(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \ .$$

We denote by $\hat{f} : \mathbb{R}^n \rightarrow \mathbb{R}$ the ideal, real-valued specification of the function that a developer may want to compute (where n is the number of function arguments, $n = 2$ for our example). While such a function can in principle be implemented exactly, e.g. using rational arithmetic, such an evaluation is generally slow. Hence, in practice, the function would be implemented in finite precision. In this paper, we consider double-precision (64 bit) IEEE 754 [18] floating-point arithmetic, which is one of the most commonly used finite precisions (though our approach generalizes to other floating-point precisions as well). We denote this finite-precision implementation by $f : \mathbb{F}^n \rightarrow \mathbb{F}$.

When evaluating f, each computed intermediate value has to be potentially rounded to a value that is representable in finite precision, introducing a *roundoff error*. While each roundoff error individually is (usually) small, the errors propagate and accumulate during the computation, resulting in potentially large errors on a function's result [20]. It is thus important to be able to make statements about this error, for instance as an absolute error: $f_{err}(\bar{x}) = |\hat{f}(\bar{x}) - f(\bar{x})|, \bar{x} \in \mathbb{F}^n$, where we assume that $\bar{x}$ are 'finite' values and not one of the Not-a-Number or Infinity special floating-point values. Our approach assumes and proves that all computations remain within the number ranges of the chosen floating-point precision and that special values never occur during expression evaluation.

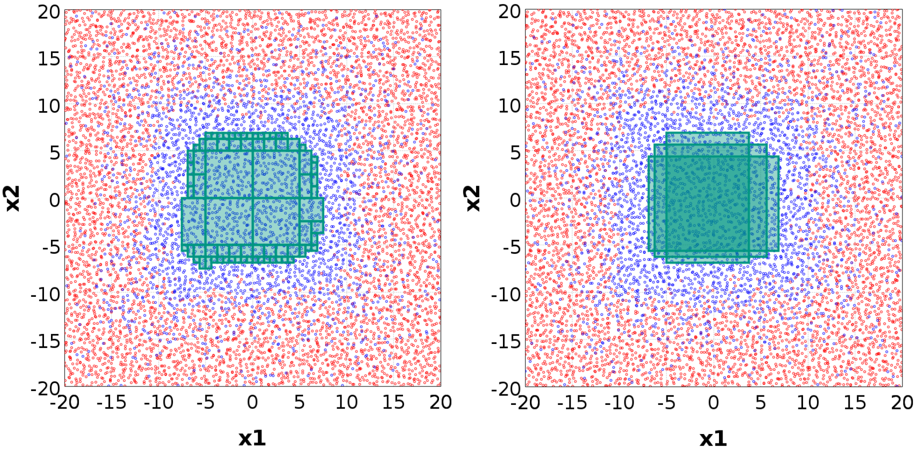
In this paper, we aim to synthesize an interval-valued precondition $\psi(\bar{x})$ that satisfies Eq. (1) ($\forall \bar{x}.\ \psi(\bar{x}) \rightarrow f_{err}(\bar{x}) \leq \varepsilon$) where ψ is of the form:

$$\bigvee_{k=1}^{m} \bigwedge_{i=1}^{n} x_i \in [a_{k,i}, b_{k,i}]$$

I.e. such a precondition represents the (set-theoretic) union of m domains of dimension n. To obtain a precondition that can be efficiently checked, we aim to keep m small (< 10), while the precondition should nonetheless be as weak as possible, i.e. cover as much of the input space as possible.

Our precondition inference starts from an initial search area which may be either specified by the user, be defined, for example, by an embedded sensor output domain, or be computed by a static analysis on the call site(s) of f. For our `himmilbeau` example, we assume $x_1, x_2 \in [-20, 20]$ as the search area, and $\varepsilon = 1.4211\text{e-}12$ as the target error bound.

In the first step, our approach samples inputs from the initial search area at random, and evaluates the function f on each input in double precision arithmetic and approximates its corresponding specification $\hat{f}$ using 128 bit arbitrary-precision arithmetic [2]. Comparing the results from the double- and higher-precision evaluations gives us an estimate of the roundoff error. We use this estimate to mark each input as valid or invalid, i.e. as satisfying or violating the postcondition, respectively. Fig. 1 shows the valid and invalid samples for our running example in blue and red, respectively. Note that the error bounds obtained from these samples do not have to be sound, as they are used only for guiding the precondition search; our technique will use static analysis to verify each precondition candidate soundly. Furthermore, the sampling also does not need to identify the exact bounds between valid and invalid samples. As Fig. 1 indicates, such bounds would lead to highly discontinuous preconditions that would be of limited practical use.



(a) Precondition with subdivision

(b) Optimized precondition

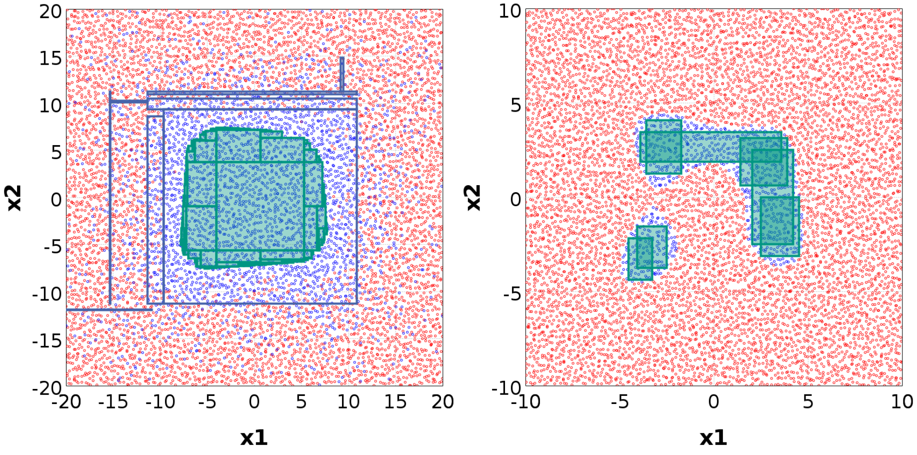
Fig. 1: The sampled `himmilbeau` function. Blue and red points indicate valid and invalid input values, respectively. The rectangles show the inferred preconditions.

Starting from these samples, we explore two techniques. First, we use *interval subdivision* to subdivide the initial search area into equal interval regions (domains such that every dimension is bounded by an interval), and then check each region individually using sound static analysis for whether it is a valid part of the precondition. Fig. 1a shows the generated precondition in green. To reduce the number of regions in the precondition for a simpler and more efficient precondition, we propose an optimization algorithm that approximates the initial verified precondition with fewer, larger regions; the result of this optimization is shown in Fig. 1b.

Subdivision may be inefficient when only a small part of the initial search area constitutes a valid precondition. We thus further explore an approach based on *classification tree learning* that starts from the valid and invalid samples and learns an initial candidate precondition, or a set of candidates if the space of valid samples is disjoint. Then, we again use static error verification to search for sound preconditions. Fig. 2a shows the generated precondition in green.

Ultimately, an inferred precondition allows us to refactor floating-point programs such that they use computations in floats if the result is known to be accurate, and resort to high-precision libraries otherwise. For example, a C-implementation of the `himmilbeau` example using the precondition from Fig. 1b, achieves a 8.6% speed-up against a pure high-precision implementation (on randomly chosen inputs from the range $[-20, 20]$). The precondition that triggers the optimization covers 11.5% of the input domain, hence the size of a precondition nearly directly translates to performance improvements.

The inferred precondition will in general be stronger than the weakest possible precondition, i.e. our inferred preconditions do not cover all of the blue points in Fig. 1 and Fig. 2. There are several reasons: The verification backend



(a) Precondition with classification tree (b) Precondition for range postcondition

Fig. 2: Inferred preconditions for `himmilbeau` using the classification tree approach for the error postcondition, and subdivision for the range postcondition.

has to rely on abstractions and can thus not always verify a valid precondition candidate. Furthermore, due to runtime considerations of our algorithm, the approaches cannot operate on arbitrarily detailed intervals.

Finally, while we discussed our precondition inference for postconditions that target an error bound, our approach equally works for postconditions that specify a target *range*, e.g. that require that the value of the result of our `himmilbeau` function is within given bounds ($f(\bar{x}) \in [-100, 100]$). We show the precondition inferred for this case using subdivision and subsequent optimization in Fig. 2b.

3 Precondition Inference by Subdivision

The first approach that we propose finds preconditions by recursively splitting the initial search area along the parameter axes until it finds interval domains for which the verification backend is able to prove that the target postcondition holds for all inputs. This approach is inspired by interval subdivision that is being used, for example, in roundoff error bound analysis to reduce the amount of over-approximations due to abstractions.

However, a naive application of subdivision for precondition inference is not practical. Each parameter's interval has to be subdivided several times in order to find verifiable preconditions, leading to a large number of regions especially for multi-variate functions. If we then run the relatively expensive verification procedure on each of these regions, the overall running time quickly becomes unreasonable. Furthermore, a precondition consisting of a large number of small interval regions is inefficient to evaluate and unwieldy. We thus combine static and dynamic verification (Sec. 3.1), and optimize the generated preconditions to yield more compact representations (Sec. 3.2).

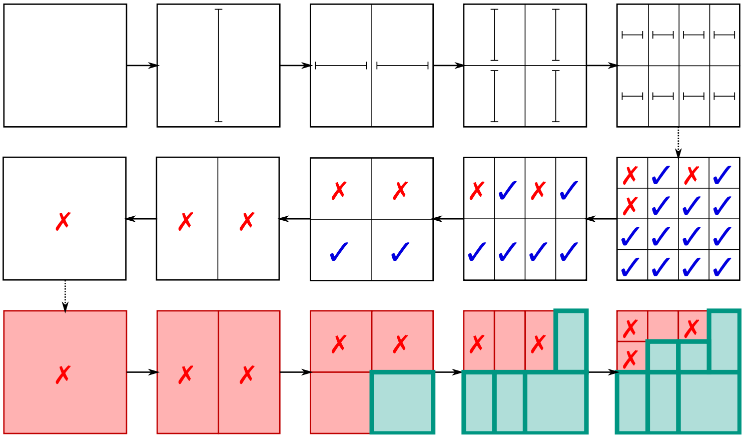


Fig. 3: Illustration of recursive subdivision in two dimensions.

Algorithm 1 Recursive Subdivision

```
given arithmetic expression $expr$, postcondition $post$
procedure EXTRACTPRE($node$)
    if $node \in valid$ then
        if verify($node.region, expr, post$) then
            return $node.region$
    if $node$ is a leaf then return $\emptyset$
    else return EXTRACTPRE($n.left$) $\cup$ EXTRACTPRE($n.right$)
```

3.1 Extracting a Verified Precondition from Subdivisions

Our approach starts by building a binary tree, where each node represents an interval region in the search area. The tree is generated by recursively splitting intervals along one parameter axis into two equally sized intervals (called *left* and *right*), splitting along each parameter axis in turn. The top part of Fig. 3 illustrates this subdivision for a two-dimensional example and with a maximum subdivision depth of 4. From left to right, the nodes are repeatedly subdivided until there are 16 leaf nodes.

Our algorithm then runs dynamic sampling (as described in Sec. 2) for each leaf node l. A node l is marked as *valid* (blue check marks in Fig. 3) if the post-condition is satisfied for *all* samples, and as *invalid* (red cross marks) otherwise. The middle part of Fig. 3 shows how these markers ascend to the root of the tree: An inner node i is marked valid if and only if both of its children are valid: $i \in valid \leftrightarrow (i.left \in valid \land i.right \in valid)$.

Next, our approach performs a recursive descent (shown in Algorithm 1) from the root node to extract the precondition. The verification backend is queried (`verify` in the algorithm) to verify that intervals are valid (sound) preconditions for all inputs in a given region. As a heurisitic, verification is attempted as close to the root of the tree as possible, as thus a single verification attempt can verify

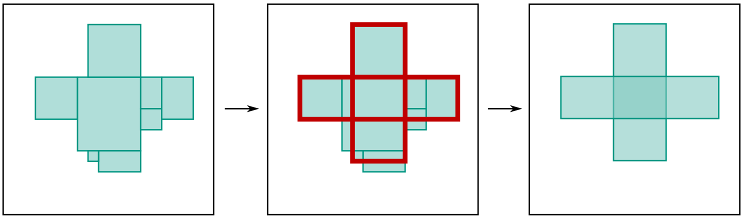


Fig. 4: Approximating generated preconditions.

a larger volume. On the other hand, the verification is more likely to fail, which may increase running time of the algorithm. Verification is futile and thus not attempted for an invalid node ($node \notin valid$). In this case, or if the verification back-end fails to verify, the procedure descends further down the tree.

The bottom part of Fig. 3 illustrates this procedure. No verification is attempted on the root node and its first degree children as they are invalid. Verification is attempted for the two valid grandchild nodes of the root that were marked with a blue check mark. For the lower right node verification is successful, so there is no need to further descend to its child nodes. Verification fails for the left one, which means it has to be subdivided again, like its two remaining sibling nodes. Sometimes subdivision is needed to verify a region even if all of it is ultimately verifiable, such as the lower left region in the last subdivision step. The reason for this is that subdivision generally reduces over-approximations due to the abstractions that the sound verification procedure relies on, and thus often allows to compute tighter error bounds [10].

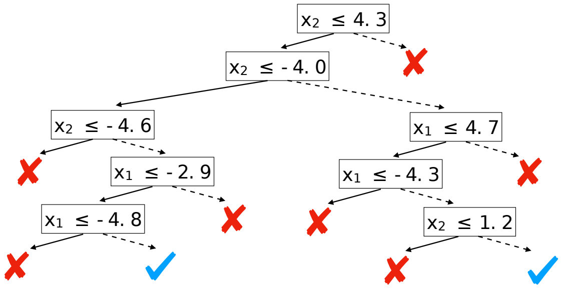
The maximum subdivision depth controls the precision of the approach. With larger depth, the generated preconditions can have a larger volume, i.e. be weaker, but this comes at the cost of a longer running time of the algorithm.

The union of all valid regions extracted from the tree is returned as a precondition. This precondition is sound, since each region has been verified by a sound roundoff error analysis.

3.2 Precondition Optimization

Depending on the subdivision depth, the number of individual regions in a generated precondition can easily reach into the thousands. We observed that one can often approximate the result with significantly fewer regions, while only marginally reducing their volume. Fig. 4 shows an example precondition generated by subdivision on the left, and the optimized precondition on the right. The precondition on the right needs only two regions instead of 8, and covers most of the originally generated precondition and is thus only slightly stronger.

Note that simply picking the largest individual interval regions from the generated precondition is in general insufficient: larger regions may be found within the verified area by composing parts of different intervals into larger ones. While one could in principle use simplification algorithms inside constraint solvers for



(a) Positive/negative decisions have solid/dashed lines. Leaves can be valid (✓) or invalid (✗).

$$\begin{gathered}(x_1 \in [-4.8, -2.9] \ \wedge \\ x_2 \in [-4.6, -4.0]) \\ \vee \\ (x_1 \in [-4.2, 4.7] \ \wedge \\ x_2 \in [1.2, 4.3])\end{gathered}$$

(b) Precondition candidate.

Fig. 5: A sample classification tree and the extracted precondition candidate

this task, such algorithms are not targeting our use-case, i.e the smallest formula that covers the biggest valid region.

Thus, we propose an optimization that starts by identifying the interval region that covers the largest verified area and that possibly (partially) covers several interval regions from the originally generated precondition. It then iteratively repeats this process and keeps adding regions that provide the most additional coverage. Since our algorithm is greedy, it is not guaranteed to find an optimal solution, but our experiments have shown that the approximation is very decent even for small numbers of representing regions. Since only regions covering verified areas are added, the optimized precondition is sound. This optimization is also fast compared to the rest of the procedure, since it does not run roundoff verification.

This precondition optimization step can be applied on preconditions obtained from both inferences approaches (recursive subdivision and the refinement approach from the upcoming section), but the effects are more pronounced for the subdivision approach as it usually produces results with more individual regions.

4 Precondition Inference by Decision Tree Learning

Our second precondition inference technique leverages the dynamic samples in a different way: it uses them to generate initial precondition candidates using decision tree learning [4], a well-known algorithm in supervised machine learning. These candidates are subsequently *refined* to obtain sound preconditions. We consider two such refinements in Sec. 4.2 and Sec. 4.3.

4.1 Extracting Candidates from a Classification Tree

First, our algorithm samples the search area as described in Sec. 2, and marks each sample as *valid* or *invalid* depending on whether or not it satisfies the postcondition. The marked or ‘classified’ samples serve as the training data to train a classification tree (CT) using decision tree learning. A CT is a binary

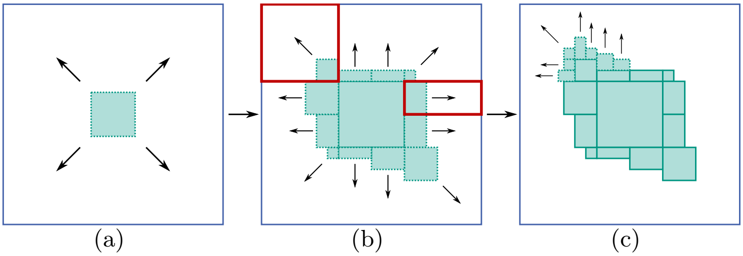


Fig. 6: Illustration of a single candidate (blue rectangle) refinement

tree in which the inner nodes are tests on the data and each leaf is labeled with a category. To classify an individual input, one follows the path given by the tests in the CT and obtains the label of the reached leaf as an answer.

We use CTs to find a simple classification that separates the *valid* from the *invalid* samples. Fig. 5a shows such a CT for our example `himmilbeau` function. Note that all tests in the CT are comparisons between a variable and a constant. From this CT, we can extract representations for the category *valid* by enumerating all paths from the root to valid leaves and collect (i.e. conjoin) all conditions (resp. their negation for negative edges). Due to the choice of simple comparisons with constants for tests, the result can be expressed as bounds on the input variables, which describes a set of interval regions. Fig. 5b shows the (simplified) precondition candidates extracted from Fig. 5a.

4.2 Refining Candidates by Growing Regions

Heuristics are applied when training CTs, and the classification has only been obtained from a set of few random samples. It is hence very likely that the candidates still contain inputs for which the desired postcondition does not hold. They need to be processed to obtain valid preconditions.

Fig. 6 illustrates our first candidate refinement process. The outer blue square represents the initial candidate. The verification backend is used to identify regions within it that verifiably are preconditions, shown as filled green rectangles in the figure. First, a small initial region in the center of the candidate is grown as much as possible without losing verifiability (Fig. 6a). When the maximal region has been found, additional precondition regions are inferred along the boundary of the region (Fig. 6b). To this end, extension candidates (two examples are shown as red rectangles) are identified as the largest possible regions to add in particular directions. The mentioned growing mechanism infers maximum regions within the extension candidates. For every added region, the extension process is repeated (Fig. 6c) until a maximum refinement depth has been reached.

Algorithm 2 shows the pseudocode procedure REFINECANDIDATE returning a verified precondition for a candidate *region*. The algorithm keeps a set M of extension candidates and searches for the largest verifiable region inside each extension candidate using BINSCALESEARCH (binary search on interval regions)

Algorithm 2 Candidate Refinement

```
given arithmetic expression $expr$, postcondition $post$, binary search depth $d$
procedure refineCandidate($region$)
    $result \leftarrow \emptyset$
    $M \leftarrow \{(\textsc{center}(region), region)\}$
    while $M \neq \emptyset$ do
        choose $(min, max) \in M$ and remove
        $verified \leftarrow \textsc{binScaleSearch}(min, max)$
        if $verified \neq \emptyset$ then
            $result \leftarrow result \cup \{verified\}$
            $M \leftarrow M \cup \textsc{genExtensionCandidates}(verified, max)$
    return $result$
```

which invokes the verification backend. The procedure center computes the center of a region used as the starting point for growing an initial solution, and genExtensionCandidates produces new extensions candidates (in form of min/max pairs of regions) to be explored.

In the implementation, the set M is realized as a priority queue favoring potential additions far from the original candidate's border that can thus grow easily, and the number of iterations is bounded by a configurable parameter.

4.3 Refining Candidates by Recursive Subdivision

Instead of this refinement approach for precondition candidates, the subdivision technique from Sec. 3 can alternatively also be applied to obtain valid preconditions from candidates. The candidate production using a CT then serves as a first step narrowing an initial search region to a smaller region in which subdivision can operate productively, in particular because a finer mesh can be applied on the interesting regions, which is better for verification with the backend verifier.

5 Evaluation

Implementation We implemented both precondition inference approaches in the open-source tool Daisy [10], building on the static range and error analyses that Daisy provides. In particular, we use Daisy's interval analysis for computing real-valued ranges and affine arithmetic for computing roundoff error bounds. We use the `DecisionTree` class from the Smile library [1] for classification tree learning. Empirically, we have identified the following default parameters that produce good results on the benchmarks on average, while not being prohibitive for larger benchmarks: we limit the maximum depth for classification tree learning to 8 and the depth for binary search during refinement in the classification tree approach to 10. When combining classification tree learning with subdivision, we limit the decision tree depth at 12. We use 8192 samples for classification tree learning and 16 samples per subdivided region for our subdivision approach.

Benchmarks We evaluate our precondition inference approaches on benchmarks from the benchmark suite FPBench [8] that is widely used in the floating-point research community. Each benchmark consists of an arithmetic expression and typically comes with a precondition specifying the input domain of the expression. For a few benchmarks where no input domain is given, we add one manually. For our evaluation, we require a postcondition to be given that specifies a target error bound or a range. Since these are not provided by FPBench as-is, we generate them for our experiments as follows. We compute error bounds and result ranges based on the existing original input domains as specified in FPBench, and use these as two separate target postconditions. We exclude benchmarks for which Daisy is not able to compute errors or ranges, e.g. because they contain conditional statements. In total, we generate a set of 99 benchmarks with postconditions specifying an error bound, and a separate set of 99 benchmarks with postconditions specifying a target range, with the following dimensionalities:

dimension	1	2	3	4	6	8	9
# benchmarks	33	29	16	4	12	1	4

Baseline In the absence of existing tools for floating-point precondition inference or the ground truth[5], we compare the preconditions inferred by our approaches against the original preconditions specified in FPBench. Indeed, the original precondition from FPBench is—by construction—a valid precondition.

We measure the quality of an inferred precondition as a relative volume, i.e. the ratio of the volume of the generated precondition over the volume of the original precondition. A relative volume greater than one is obtained if the original domain specification is strong and the approaches discover valid preconditions beyond the original specification. For many benchmarks, however, obtaining a relative volume close to one is close to the optimal result. (Measuring the absolute volumes is not meaningful as they are highly benchmark dependent.)

Setup Our techniques rely on an initial search area provided by the user. While it may be convenient if our algorithms considered an unbounded initial space, i.e. all possible floating-point values, this is practically infeasible. The valid precondition typically covers only a very small part of this 'unbounded' domain, and it would thus be computationally very expensive to search for.

For our evaluation, we consider two sets of initial search areas: We use the original domain specified in FPBench *scaled* uniformly around their centers to contain 100 times the original volume, and we use a large *fixed* initial domain for all benchmarks bounding all input arguments in $[-10^8, 10^8]$. For both initial search areas, it is unlikely that the entire area would be a valid precondition.

Comparison of Approaches Simply comparing the relative volume of the preconditions does not consider that each approach would be able to produce bigger preconditions by investing more computational effort. Conversely, the running

[5] The exact ground truth would be highly discontinuous, and would require sampling of all floating-point inputs, which is infeasible for double precision.

precondition:	error			range		
	TO	fail	best	TO	fail	best
scaled search area						
subdivision	14	2	67	10	2	56
tree refinem.	6	3	22	9	3	24
hybrid	6	2	11	5	9	23
fixed search area						
subdivision	333	57	22	312	80	8
tree refinem.	344	73	4	319	81	4
hybrid	344	56	8	320	79	3

Fig. 7: Summary statistics

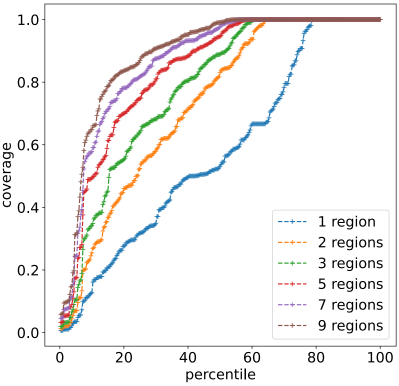


Fig. 8: Cactus plot evaluating the precondition optimization

times cannot be compared in isolation. Thus, we compare the relative volumes of generated preconditions per invested time[6]. We use a timeout of 20 minutes for each benchmark and parameter setting.

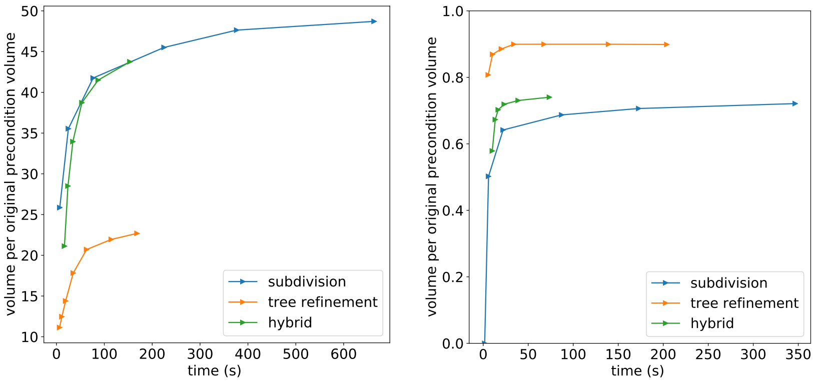
We consider our effectively three approaches: *subdivision*, *tree refinement* (with growing candidates), and tree refinement with subdivision, that we call *hybrid* for the sake of this evaluation. For this comparison, we initially do not use the precondition optimization from Sec. 3.2, and evaluate it separately. We observe that for the subdivision and the hybrid approach, the maximum depth of the subdivision tree significantly affects the running time of the algorithm. For the tree refinement, the most relevant parameter is the number of refinement candidates considered for the growing-based refinement. We thus vary these parameters and keep all others to the default values given in Sec. 5. In total, we run 3762 experiments using the scaled and 1782 experiments using the fixed search area.

Fig. 7 summarizes our results. 'TO' counts the number of times an individual run timed out. 'Fail' means that no precondition was found by a search strategy for any of the tested parameters. 'Best' counts the number of benchmarks for which an approach was able to find the best (weakest) precondition (with any parameter setting); when the numbers do not add up to 99, it is due to ties.

Clearly, our precondition inference is more effective for the scaled search area benchmarks; it is able to find preconditions in nearly all runs. However, it is able to find some preconditions even for the very large area, where the verifiable regions are often vanishingly small. Also, we observe that no one approach is universally better than the others, as each is best on some set of benchmarks.

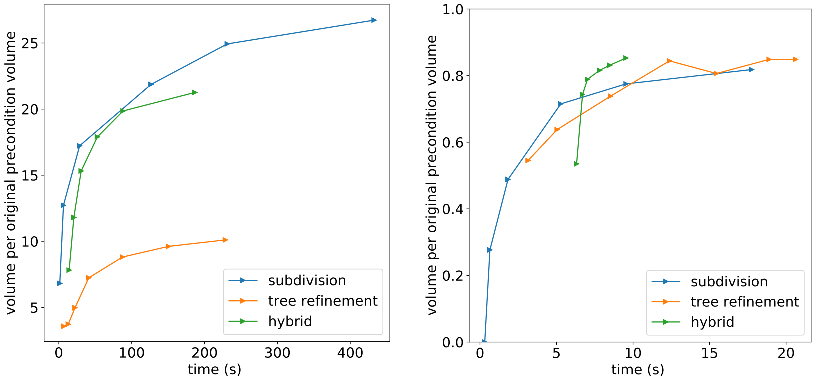
Fig. 9 visualizes the relative volumes of generated preconditions by the different approaches per running time of the algorithm, for benchmarks where the postconditions bound the roundoff error and for the scaled input search areas. Each point corresponds to one parameter setting. Fig. 9a averages over all benchmarks, whereas Fig. 9b averages only over benchmarks where the gener-

[6] We ran all experiments on a Mac mini with an 6-core Intel i5 processor at 3 GHz with 16 GB RAM running macOS Catalina.



(a) Average over all benchmarks (b) Average for small preconditions

Fig. 9: Comparison of approaches without optimization: average relative volume per time (seconds) for error postconditions and 100x search area.



(a) Average over all benchmarks (b) Average for small preconditions

Fig. 10: Comparison of approaches without optimization: average relative volume per time (seconds) for range postconditions and 100x search area.

ated precondition was small, i.e. at most 1.2 times the original precondition. We show the analogous plots for the range postconditions in Fig. 10.

We observe that averaged over all benchmarks, the subdivision and hybrid approaches perform significantly better than the tree refinement approach. In fact, our techniques are able to identify preconditions that are, on average, significantly larger than the original precondition. If we consider only those 33 benchmarks, where only a relative small precondition was generated, we see that tree refinement shows the, on average, best benefit. For our range benchmarks (Fig. 10), we observed on average a slight benefit of the hybrid approach for small preconditions. Note that even when 'small' preconditions are generated, they nearly cover the entire input search area, i.e. our precondition inference is able to recover most of the original preconditions.

Precondition Optimization Finally, we evaluate the effectiveness of the precondition optimization on the subdivision and hybrid approach (we have not observed the optimization to be particularly useful for tree refinement). For this evaluation, we fix a particular parameter setting that achieves a good trade-off between relative volume of inferred preconditions and running time of inference. Then we vary the number of target regions that the optimization should produce. On average, the preconditions generated for this experiment consisted of 120 distinct regions *before* optimization. For each run, we compute the *coverage* of the optimized precondition, i.e. the ratio of the optimized over the non-optimized inferred precondition. Fig. 8 visualizes the results of this experiment as a cactus plot where we sort the runs by coverage. For example, the value 0.27 for 1 region at the 20th percentile means that in 80% of the runs, the coverage of the optimized precondition was at least 0.27. As expected, the more regions are allowed, the better the coverage of the optimized preconditions becomes. Overall, we see that our inference with optimization is able to generate relatively simple preconditions (i.e. with just a few regions) in reasonable time that nonetheless cover large parts of the verifiable area for many of the benchmarks.

Case Study We demonstrate the benefits of our precondition inference on a practical problem that inspired this work. We consider the 9-dimensional function to calculate the scalar triple product $\alpha \cdot (\beta \times \gamma)$ of three 3-dimensional vectors $\alpha, \beta, \gamma \in \mathbb{R}^3$, based on the requirements of an assumed use case: each parameter will be within a range of $[-1337, 1337]$, and we require the error of the result to be at most $3 \cdot 10^{-6}$. This use case arose in a convex hull algorithm for scientific computing in material sciences.

Running the recursive subdivision approach for this expression with a subdivision depth of 14 and 262144 samples yields the following results: In roughly 13 minutes, the approach produces a precondition that covers about 67 percent of the search area and consists of 4608 individual intervals. In another 112 seconds, the optimization algorithm produces a precondition consisting of only two intervals which together cover 51% of the verified area and 34% of the search area. Using this optimized precondition, we can create a hybrid implementation of the original function, which decides whether to use a (exact) rational or floating-point version dynamically. Even with the added overhead from checking the precondition, the required runtime reduces from $17.13s$ for a purely rational implementation to $10.77s$ for the hybrid implementation for running the function 100000 times with random inputs from the input space. A similar speedup can be observed when using a higher precision floating-point implementation instead of an exact rational implementation in case the precondition does not hold.

6 Related Work

The precondition synthesis approaches presented in this work rely on state-of-the-art floating-point verification and analysis tools to verify precondition candidates and guarantee their soundness. While we have used the Daisy framework

[10] as a verification backend, any tool able to calculate sound bounds for errors or result ranges of floating-point functions could be used instead: Fluctuat [14], Gappa [11], FPTaylor [29], Real2Float [21] and PRECiSA [22].

We are not aware of an existing technique that can generate sound preconditions for floating-point functions. The closest related techniques are optimizations that identify certain parts of the input domain, for which a rewriting of the input program results in a smaller roundoff error [26,32,30]. These rewritings are based on real-valued identities, leveraging the fact that floating-point arithmetic is e.g. not associative, or polynomial approximations. The split of the input domain can be viewed as a kind of precondition, however, the goal and guarantees provided are very different. The aim is to identify and repair large roundoff errors, whereas our approach tries to identify the input domain with reasonable errors. Furthermore, all of the techniques rely on dynamic analysis and thus do not provide soundness guarantees.

Dynamic analysis is frequently being used to estimate the magnitude of roundoff errors [3], and several works have developed a targeted search towards inputs that cause particularly large errors [31,6,33], in order to identify worst-case errors. Our precondition inference combines dynamic and static analysis in a novel way in that the dynamic analysis serves a pre-processing step to explore the input domain. As such, the goal of our dynamic analysis is different from existing ones, as we want it to explore the input domain evenly, instead of focusing on a (possibly small) part of the input domain with large errors.

One possible use of our inferred preconditions is to be able to generate implementations that choose an efficient floating-point precision whenever possible, and otherwise use some ‘safe’ higher precision. In that, our approach is related to mixed-precision tuning techniques that mostly focus on implementations that mix single, double and quad floating-point precision. Some of these use dynamic analysis to estimate errors and thus do not provide sound guarantees [25,19,17,15], and others use static analysis with accuracy guarantees, but less scalability [5,9]. Mixed-precision tuning generally works well when the target error bounds are close to the error bounds of uniform-precision implementations [9,27]. We consider mixed-precision tuning complementary to our precondition inference; for instance, preconditions generated by our approaches could be used as a starting-point for mixed-precision tuning.

7 Conclusion

We have presented the first precondition inference techniques from floating-point accuracy and range postconditions, using a combination of dynamic and static analysis. Each of the three approaches that we explored generate good results from reasonably sized initial search areas with acceptable computational effort and have different strengths and weaknesses; neither approach is universally better than the others. One of the main challenges for future work is to improve the identification of preconditions when the initial search areas are very large, which we have identified as a particular challenge.

References

1. Smile - Statistical Machine Intelligence and Learning Engine, https://haifengl.github.io/
2. The GNU MPFR Library (2020), https://www.mpfr.org/
3. Benz, F., Hildebrandt, A., Hack, S.: A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In: Programming Language Design and Implementation (PLDI) (2012). https://doi.org/10.1145/2254064.2254118
4. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and Regression Trees. CRC press (1984)
5. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous Floating-Point Mixed-Precision Tuning. In: Principles of Programming Languages (POPL) (2017). https://doi.org/10.1145/3009837.3009846
6. Chiang, W., Gopalakrishnan, G., Rakamaric, Z., Solovyev, A.: Efficient Search for Inputs Causing High Floating-Point Errors. In: Principles and Practice of Parallel Programming (PPoPP) (2014). https://doi.org/10.1145/2555243.2555265
7. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: Guessing Formal Specifications Using Testing. In: Tests and Proofs (2010). https://doi.org/10.1007/978-3-642-13977-2_3
8. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In: Numerical Software Verification (2017). https://doi.org/10.1007/978-3-319-54292-8_6
9. Darulova, E., Horn, E., Sharma, S.: Sound Mixed-Precision Optimization with Rewriting. In: International Conference on Cyber-Physical Systems (ICCPS) (2018). https://doi.org/10.1109/ICCPS.2018.00028
10. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018). https://doi.org/10.1007/978-3-319-89960-2_15
11. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Transactions on Mathematical Software **37**(1) (2010). https://doi.org/10.1145/1644001.1644003
12. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM **18**(8) (1975). https://doi.org/10.1145/360933.360975
13. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming **69**(1-3) (2007). https://doi.org/10.1016/j.scico.2007.01.015
14. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011). https://doi.org/10.1007/978-3-642-18275-4_17
15. Guo, H., Rubio-González, C.: Exploiting Community Structure for Floating-Point Precision Tuning. In: International Symposium on Software Testing and Analysis (ISSTA) (2018). https://doi.org/10.1145/3213846.3213862
16. Himmelblau, D.M., Clark, B.J., Eichberg, M.: Applied Nonlinear Programming. McGraw-Hill (1972)
17. Ho, N., Manogaran, E., Wong, W., Anoosheh, A.: Efficient Floating Point Precision Tuning for Approximate Computing. In: Asia

and South Pacific Design Automation Conference (ASP-DAC) (2017). https://doi.org/10.1109/ASPDAC.2017.7858297

18. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
19. Lam, M.O., Vanderbruggen, T., Menon, H., Schordan, M.: Tool Integration for Source-Level Mixed Precision. In: Workshop on Software Correctness for HPC Applications (Correctness) (2019). https://doi.org/10.1109/Correctness49594.2019.00009
20. Loh, E., Walster, G.W.: Rump's Example Revisited. Reliable Computing **8**(3) (2002). https://doi.org/10.1023/A:1015569431383
21. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. ACM Transactions on Mathematical Software **43**(4) (2017). https://doi.org/10.1145/3015465
22. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: Computer Safety, Reliability, and Security (SAFECOMP) (2017). https://doi.org/10.1007/978-3-319-66266-4_14
23. Moy, Y.: Sufficient Preconditions for Modular Assertion Checking. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2008). https://doi.org/10.1007/978-3-540-78163-9_18
24. Muller, J.M., Brunie, N., de Dinechin, F., Jeannerod, C.P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-76526-6
25. Nathan, R., Naeimi, H., Sorin, D.J., Sun, X.: Profile-Driven Automated Mixed Precision. CoRR **abs/1606.00251** (2016), http://arxiv.org/abs/1606.00251
26. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically Improving Accuracy for Floating Point Expressions. In: Programming Language Design and Implementation (PLDI) (2015). https://doi.org/10.1145/2737924.2737959
27. Rabe, R., Izycheva, A., Darulova, E.: Regime Inference for Sound Floating-Point Optimizations. ACM Trans. Embed. Comput. Syst. (EMSOFT) **20**(5s) (2021). https://doi.org/10.1145/3477012
28. Seghir, M.N., Kroening, D.: Counterexample-Guided Precondition Inference. In: Programming Languages and Systems (ESOP) (2013). https://doi.org/10.1007/978-3-642-37036-6_25
29. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. ACM Transactions on Programming Languages and Systems **41**(1) (2018). https://doi.org/10.1145/3230733
30. Wang, X., Wang, H., Su, Z., Tang, E., Chen, X., Shen, W., Chen, Z., Wang, L., Zhang, X., Li, X.: Global Optimization of Numerical Programs via Prioritized Stochastic Algebraic Transformations. In: International Conference on Software Engineering (ICSE) (2019). https://doi.org/10.1109/ICSE.2019.00116
31. Xia, Y., Guo, S., Hao, J., Liu, D., Xu, J.: Error Detection of Arithmetic Expressions. The Journal of Supercomputing (2020). https://doi.org/10.1007/s11227-020-03469-7
32. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. Proceedings of the ACM on Programming Languages **3**(POPL) (2019). https://doi.org/10.1145/3290369

33. Zou, D., Wang, R., Xiong, Y., Zhang, L., Su, Z., Mei, H.: A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In: International Conference on Software Engineering (ICSE) (2015). https://doi.org/10.1109/ICSE.2015.70

NeuReach: Learning Reachability Functions from Simulations[⋆]

Dawei Sun(✉) and Sayan Mitra

University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
{daweis2,mitras}@illinois.edu

Abstract. We present NeuReach, a tool that uses neural networks for predicting reachable sets from executions of a dynamical system. Unlike existing reachability tools, NeuReach computes a *reachability function* that outputs an accurate over-approximation of the reachable set for *any* initial set in a parameterized family. Such reachability functions are useful for online monitoring, verification, and safe planning. NeuReach implements empirical risk minimization for learning reachability functions. We discuss the design rationale behind the optimization problem and establish that the computed output is probably approximately correct. Our experimental evaluations over a variety of systems show promise. NeuReach can learn accurate reachability functions for complex nonlinear systems, including some that are beyond existing methods. From a learned reachability function, arbitrary reachtubes can be computed in milliseconds. NeuReach is available at https://github.com/sundw2014/NeuReach.

Keywords: Reachability analysis · Data-driven methods · Machine learning

1 Introduction

Reachability has traditionally been a fundamental building block for verification, monitoring, and prediction, and it is finding ever-expanding set of applications in control of cyber-physical and autonomous systems [19,23]. Reachtubes cannot be computed exactly for general hybrid models, but remarkable progress over the past two decades have led to approximation algorithms for nonlinear and very high-dimensional linear models (See, for example, [11,18,5,3,25,12,1,26,34]). All of these algorithms and tools compute the reachtube from scratch, every time the algorithm is invoked for a new initial set $\mathcal{X}_0$, even if the system model does not change. This is a missed opportunity in amortizing the cost of reachability over multiple invocations. All the applications mentioned above, like verification, monitoring, and prediction, indeed use multiple reachtubes of the same system, but from different initial sets.

[⋆] The authors were supported by research grants from the National Security Agency's Science of Security (SoS) program and National Science Foundation's Formal Methods in the Field (FMITF) program.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 322–337, 2022.
https://doi.org/10.1007/978-3-030-99524-9_17

In this paper, we present NeuReach, a tool that learns a *reachability function* from executions of dynamical systems. With the learned reachability function, for every new initial set a corresponding reachtube can be computed quickly. To use NeuReach, the user has to implement a simulator function of the underlying dynamical (or hybrid) system for generating trajectories, and several other functions for sampling initial sets. As output, the tool will generate a function which can be serialized and stored for repeated use. This function takes as input a query which is an initial set $\mathcal{X}_0$ and a time instant t, and outputs an ellipsoid, which is guaranteed to be an accurate over-approximation of the actual reachable set.

Formally, NeuReach solves a probabilistic variant of the well-studied reachability problem: the problem is to compute a *reachability function* $R(\cdot, \cdot)$ for a given model (or simulator), such that for *any* initial set $\mathcal{X}_0$ and time t, the output of the function $R(\mathcal{X}_0, t)$ is an over-approximation of the actual reachset from $\mathcal{X}_0$ at time t. That is, R is computed once and for all—possibly with an expensive algorithm—and thereafter, for every new initial set $\mathcal{X}_0$ and time t, the reachset over-approximation $R(\mathcal{X}_0, t)$ is computed simply by calling R. Thus, it enables online and even real-time applications of reachset approximations.

NeuReach computes reachability functions using machine learning. We view this as a statistical learning problem where samples of the system's trajectories have to be used to learn a parameterized reachability function $R_\theta(\cdot, \cdot)$. Because the trajectory samples are the only requirements from the underlying dynamical system to run NeuReach, it can be applied to systems with or without analytical models. In this paper, we discuss how the above problem can be cast as an optimization problem. This involves carefully designing a loss function that penalizes error and conservatism of the reachability function. With this loss function, it becomes possible to solve the problem using empirical risk minimization and stochastic gradient descent. For the sake of justifying our design, we derive a theoretical guarantee on the sample complexity using standard statistical learning theory tools.

We evaluate NeuReach on several benchmark systems and compare it with DryVR [21] which also uses machine learning for single-shot reachset computations. Results show that, with the same training data, NeuReach generates more accurate and tighter reachsets. Using NeuReach we are able to check the key safety properties of the challenging F-16 benchmark presented in [28]. To our knowledge, this is the first successful verification of at least some scenarios in this benchmark. Furthermore, as expected, once $R(\cdot, \cdot)$ is computed, it can be invoked to rapidly compute reachsets for arbitrary $\mathcal{X}_0$ and t. For example, estimating a reachset for an 8-dimensional dynamical system with an NN-controller only takes $\sim$ 0.3 milliseconds. This makes NeuReach attractive for online and real-time applications.

Contributions. (1) We present a simple but effective and useful machine-learning algorithm for learning reachability functions from simulations. With the learned reachability function, accurate over-approximation of the reachable set for *any* initial set in a parameterized family can be quickly computed, which enables real-time safety check and online planning; (2) We derive a probably approximately correct (PAC) bound on the error of the learned reachability function (Theorem 1) using techniques in statistical learning theory; (3) We evaluate the proposed tool on several benchmark dynamical systems and compare it with another data-driven reachability tool. Experiments show that NeuReach can learn more accurate and tighter reachability functions for complex nonlinear and hybrid systems, including some that are beyond existing methods.

2 Related work

Reachability analysis for models with known dynamics. This category of approaches consider the reachability analysis of models with known dynamics (i.e., white-box models). This is an active research area, and there is an extensive body of theory and tools on this topic [11,2,15,5,25,33,27,16,38,10,39]. Reachability analysis is hard in general. Exact reachability is undecidable even for deterministic linear and rectangular models [29,24]. For dynamical models described with ordinary differential equations (ODE), Hamilton–Jacobi–Bellman (HJB) equations can be used to derive the exact reachable sets [30,6,7]. An HJB equation is a partial differential equation (PDE). Solutions of this PDE defines the reachabiltiy of the underlying dynamical system. However, solving HJB equations is difficult, and such approaches do not scale to high-dimensional systems. In practice, the exact reachable set might be unnecessary. For example, over-approximations of the reachable sets could suffice for safety check purpose. To this end, many approaches and tools have been developed. For example, Flow* [11] uses the technique of Taylor model integration to compute over-approximations of the solution of an ODE.

Another series of work [22,20] leverage the sensitivity analysis of ODE to bound the discrepancy of solutions starting from a small initial set, and thus can compute an over-approximation of the exact reachable set. In [12], a Lagrangian-based algorithm is proposed, which makes use of the Cauchy-Green stretching factor derived from an over-approximation of the gradient of the solution-flows of an ODE. All of the above approaches consider set-based reachability analysis.

Data-driven reachability analysis. In the cases where the exact dynamics of the systems is unknown or partially known, the above approaches cannot be applied. One straight-forward direction is to learn the reachability from behaviors [42] of the dynamical system. Several approaches have been proposed for reachability *only* using simulations of the underlying system. These approaches include scenario optimization [14,44], sensitivity analysis [21], Gaussian processes [13], adversarial sampling [32,9], etc.

NeuReach falls in the category of approaches that use randomized algorithms for reachability analysis of deterministic (and not stochastic) systems. Another member in this category is the scenario optimization approach presented in [14]. Different from NeuReach, this method learns a single reachset for a fixed initial set and time interval instead of a mapping from arbitrary initial sets and time to the reachsets. Another approach based on scenario optimization is presented in [44]. This method computes a fixed-width reachtube by learning a function of time to represent the central axis of the reachtube. Moreover, it uses polynomials with handcrafted feature vectors for learning, which requires case-by-case design and fine-tuning. In contrast, our method learns a more flexible reachability function using neural networks and avoids the use of handcrafted feature vectors. DryVR [21] computes the reachtubes based on sensitivity analysis. It first learns a sensitivity function with theoretical guarantees, and then uses it to compute the reachset. Among all these tools or methods, we found that DryVR is the only one that has a publicly available implementation. Thus, we compared NeuReach with DryVR.

Neural networks for reachability analysis. Applications of machine learning with neural networks for reachability and monitoring has become an active research area. The approach in [23] aims to learn the reachtube from data using neural networks, with a focus in motion planning. Unlike NeuReach, this approach learns the dynamics of the reachtube, and the reachtube can be obtained by integrating that dynamics. In [30,7], neural networks are used as a PDE solver to approximate the solution of HJB equations. The approach in [36] makes use of neural networks to approximate the reachability of dynamical systems with control input. In [38,10], the authors develop a framework for runtime predictive monitoring of hybrid automata using neural networks and conformal prediction.

3 Problem setup and an overview of the tool

NeuReach works with deterministic dynamical systems. The state of the system is denoted by $x \in \mathcal{X} \subseteq \mathbb{R}^n$. We assume that we have access to a simulator function $\xi : \mathcal{X} \times \mathbb{R}_{\geq 0} \mapsto \mathcal{X}$ that generates trajectories of the system up to a time bound T. That is, given an initial state $x_0 \in \mathcal{X}$ and a time instant $t \in [0, T]$, $\xi(x_0, t)$ is the state at time t.[1]

Consider the evolution of the system from a set of initial states (*initial set*) $\mathcal{X}_0 \subset \mathcal{X}$. Lifting the notation of ξ to sets, we write the *reachset* from $\mathcal{X}_0$ as $\xi(\mathcal{X}_0, t) := \cup_{x_0 \in \mathcal{X}_0} \xi(x_0, t)$. In general, $\xi(\mathcal{X}_0, t)$ cannot be computed precisely, and thus, we resort to over-approximations of $\xi(\mathcal{X}_0, t)$ which are usually sufficient for verification and monitoring of safety and general temporal logic requirements, and also for planning. Beyond computing over-approximations of $\xi(\mathcal{X}_0, t)$ for a single $\mathcal{X}_0$ and t, we are interested in finding a *reachability function* $R : 2^{\mathcal{X}} \times [0, T] \mapsto 2^{\mathcal{X}}$ such that, ideally, $\xi(\mathcal{X}_0, t) \subseteq R(\mathcal{X}_0, t)$ for all valid $\mathcal{X}_0$ and t. NeuReach implements a solution to this problem which provides a probabilistic version of the above guarantee with some restrictions on the shape of the initial set $\mathcal{X}_0$.

In order to discuss the error of a reachability function R, we have to assume that its arguments $\mathcal{X}_0$ and t are independently chosen according to some dis-

[1] For the sake of simplicity, here we ignore issues arising from quantization and numerical errors in simulators. Such issues have been extensively studied in the numerical analysis and we refer the reader to [17] for a discussion related to verification.

tributions P_1 and P_2, i.e. $\mathcal{X}_0 \sim P_1$ and $t \sim P_2$. Also, we need a distribution function $\mathcal{D}(\cdot)$ such that $\mathcal{D}(\mathcal{X}_0)$ is distribution over $\mathcal{X}_0$. For example, $\mathcal{D}(\mathcal{X}_0)$ could be the uniform distribution over $\mathcal{X}_0$. Given these distributions, the *error* of a reachability function is defined as:

$$\Pr_{\mathcal{X}_0 \sim P_1, t \sim P_2, x_0 \sim \mathcal{D}(\mathcal{X}_0)} \left[\xi(x_0, t) \notin R(\mathcal{X}_0, t) \right]. \tag{1}$$

Here, we assume that the joint distribution of $(\mathcal{X}_0, t, x_0)$ is defined on the Borel $\sigma-$algebra such that any Borel set is measurable. Given the fact that R is continuous[2] and ξ as the trajectory of a dynamical system is at least piece-wise continuous, the set of all tuples $(\mathcal{X}_0, t, x_0)$ that satisfy $\xi(x_0, t) \notin R(\mathcal{X}_0, t)$ must be a Borel set, and thus is measurable. Therefore, the above probability is well defined.

User interface and data representation. P_1, P_2 and $\mathcal{D}$ are specified by the user as functions generating samples (explained below). The input and output of the reachability function $R(\mathcal{X}_0, t)$ involve infinite objects, and in order to learn R, first, we need some finite representations of these objects. In NeuReach, $\mathcal{X}_0$ is picked from a user-specified family of sets where each set can be represented by a finite number of parameters. For example, $\mathcal{X}_0$ could be a ball and represented by two parameters — center and radius. From here on, we will not distinguish between $\mathcal{X}_0$ and its parameterized representation. Similarly, the reachset $R(\mathcal{X}_0, t)$ also needs a representation. NeuReach represents the reachsets with ellipsoids. Given a vector $x_0 \in \mathbb{R}^n$ and a matrix $C \in \mathbb{R}^{n \times n}$, the set $\mathcal{E}(x_0, C) := \{x \in \mathbb{R}^n : \|C \cdot (x - x_0)\|_2 \leq 1\}$ is an *ellipsoid*. Thus, given the center, an ellipsoid can be represented by an $n \times n$ matrix.

In order to use NeuReach, the user has to implement the following functions.

(i) `sample_X0()`: Produces a random initial set $\mathcal{X}_0$ from a distribution P_1. Specifically, the parameterized representation of $\mathcal{X}_0$ is returned.
(ii) `sample_t()`: Produces a random sample of t from a distribution P_2.
(iii) `sample_x0(X0)`: Takes an initial set $\mathcal{X}_0$, and produces a random sample of $x_0 \in \mathcal{X}_0$ according to a distribution $\mathcal{D}(\mathcal{X}_0)$.
(iv) `simulate(x0)`: Takes an initial state x_0 and generates a finite trajectory $\xi(x_0, \cdot)$ which is a sequence of states at some time instants. The user should make sure that for every time instant returned by `sample_t()`, a state corresponding to it can be found in the simulated trajectory.
(v) `get_init_center(X0)`: Takes an initial set $\mathcal{X}_0$ and returns $\mathbb{E}\left[\mathcal{D}(\mathcal{X}_0)\right] := \mathbb{E}_{x \sim \mathcal{D}(\mathcal{X}_0)}[x]$, which is the mean value of x over the initial states.

Given these functions, NeuReach computes a reachability function R with an error guarantee (Theorem 1). The reachset $R(\mathcal{X}_0, t)$ is an ellipsoid centered at $\xi(\mathbb{E}\left[\mathcal{D}(\mathcal{X}_0)\right], t)$. As the output, NeuReach will generate a Python function `R(X0, t)`. This function can be serialized and stored on disk for future use. When calling this function, the user provides the initial set $\mathcal{X}_0$ and t, and then an $n \times n$ matrix representing the shape of the ellipsoid will be returned.

[2] As will be stated later, R is a neural network, which is indeed continuous.

4 Design of NeuReach: Learning reachability functions

We present the design rationale behind NeuReach and discuss the learning algorithm it implements. We show that standard results in statistical learning theory give a probabilistic guarantee on the error of the learned reachability function.

4.1 Reachability with Empirical Risk Minimization

The basic idea is to model the reachset $R(\mathcal{X}_0, t)$ as an ellipsoid around $\xi(\mathbb{E}[\mathcal{D}(\mathcal{X}_0)], t)$. As stated earlier, given the center, an n-dimensional ellipsoid can be represented by an $n \times n$ matrix. Thus, learning the set-valued reachability function $R(\mathcal{X}_0, t)$ becomes the problem of learning a matrix-valued function $C(\mathcal{X}_0, t)$ that describes the shape of the set. We represent function C using parametric models, such as neural networks. Let us denote this parametric, matrix-valued function by C_θ, where $\theta \in \mathcal{W} \subseteq \mathbb{R}^p$ is the vector of parameters. The parameter θ could be, for example, a scalar representing a coefficient of a polynomial, a vector representing weights of a neural network, etc. Thus, the parametric reachability function is:

$$R_\theta(\mathcal{X}_0, t) := \mathcal{E}(\xi(\mathbb{E}[\mathcal{D}(\mathcal{X}_0)], t), C_\theta(\mathcal{X}_0, t)). \tag{2}$$

To simplify the notations, for $X = (\mathcal{X}_0, t, x_0)$ and parameter θ, we define a function $g_\theta(X) := \|C_\theta(\mathcal{X}_0, t)(\xi(x_0, t) - \xi(\mathbb{E}[\mathcal{D}(\mathcal{X}_0)], t))\|_2$. For a particular sample X and a parameter θ, if $g_\theta(X) \le 1$, then $\xi(x_0, t) \in R_\theta(\mathcal{X}_0, t)$, otherwise it is outside and contributes to the error. The goal of our learning algorithm is to find a θ to minimize the error of the resulting reachability function R_θ, which gives the following optimization problem:

$$\begin{aligned}
\theta^* &= \arg\min_\theta \Pr_{\mathcal{X}_0 \sim P_1, t \sim P_2, x_0 \sim \mathcal{D}(\mathcal{X}_0)} [\xi(x_0, t) \notin R_\theta(\mathcal{X}_0, t)] \\
&= \arg\min_\theta \mathop{\mathbb{E}}_{\mathcal{X}_0, t, x_0} \left[\mathbb{I}\left(\left\| C_\theta(\mathcal{X}_0, t) \cdot \Big(\xi(x_0, t) - \xi(\mathbb{E}[\mathcal{D}(\mathcal{X}_0)], t) \Big) \right\|_2 > 1 \right) \right] \\
&= \arg\min_\theta \mathop{\mathbb{E}}_{X := (\mathcal{X}_0, t, x_0)} [\mathbb{I}(g_\theta(X) - 1 > 0)],
\end{aligned}$$

where $\mathbb{I}(\cdot)$ is the indicator function.

In order to solve the above optimization problem using empirical risk minimization, we consider the following setup. First, a training set is constructed. We denote a training set with N samples by $S = \{X_i\}_{i=1}^N$, where the samples $X_i = (\mathcal{X}_0^{(i)}, t^{(i)}, x_0^{(i)})$ are independently drawn from the data distribution defined by $\mathcal{X}_0 \sim P_1, t \sim P_2, x_0 \sim \mathcal{D}(\mathcal{X}_0)$. The *empirical loss* on S for a parameter θ is

$$L_{ERM}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(g_\theta(X_i) - 1), \tag{3}$$

where $\ell(x) := \max\{0, \frac{x}{\alpha} + 1\}$ is the hinge loss function with the hyper-parameter $\alpha > 0$, which is a soft proxy for the indicator function. Therefore, the empirical loss L_{ERM} is a soft, empirical proxy of the actual error as defined in Equation (1).

Arguments	Default Value	Description
`system`	-	Name of the Python file containing the model.
`lambda`	0.03	λ in Eq. (4).
`alpha`	0.001	α in Eq. (3).
`N_X0`	100	$N_{\mathcal{X}_0}$: Number of initial sets.
`N_x0`	10	N_{x_0}: Number of initial states.
`N_t`	100	N_t: Number of time instants.
`layer1`	64	L_1: Number of neurons in the first layer of the NN.
`layer2`	64	L_2: Number of neurons in the second layer of the NN.
`epochs`	30	Number of epochs for training.
`lr`	0.01	Learning rate.

Table 1: Command-line arguments passed to the tool.

In addition to minimizing the empirical loss, we would also like the over-approximation of the reachset to be as tight as possible. Thus, the volume of the ellipsoid should be penalized. Inspired by [14], we use $-\log(\det(C^\intercal C))$ as a proxy of the volume of an ellipsoid $\mathcal{E}(x_0, C)$, and the following regularization term is added to penalize large ellipsoids.

$$L_{REG}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log\left(\det\left(C_\theta(\mathcal{X}_0^{(i)}, t^{(i)})^\intercal C_\theta(\mathcal{X}_0^{(i)}, t^{(i)})\right)\right).$$

Combining the two terms, we define the overall optimization problem:

$$\hat{\theta} = \arg\min_\theta L_{ERM}(\theta) + \lambda L_{REG}(\theta), \tag{4}$$

where λ is a hyper-parameter balancing two loss terms.

Machine learning setup. The training set is constructed as follows. First, we sample $N_{\mathcal{X}_0}$ initial sets by calling `sample_X0()`. Then, for each initial set, we sample N_{x_0} initial states from it using `sample_x0(X0)` and then get N_{x_0} trajectories by calling `simulate(x0)`. Finally, for each trajectory, we sample N_t time instants by calling `sample_t()`. Thus, the resulting training set contains $N := N_{\mathcal{X}_0} \times N_{x_0} \times N_t$ samples, but generating such a training set only needs $N_{\mathcal{X}_0} \times N_{x_0}$ trajectory simulations. NeuReach implements the optimization problem of Equation (4) in Pytorch [37] and solves it with stochastic gradient descent. By default, a three-layer neural network is used to represent C_θ. For n-dimensional reachsets, the number of neurons in each layer are L_1, L_2, and n^2, where L_1 and L_2 can be specified by the user. The output vector of the neural network is then reshaped to be an $n \times n$ matrix. By default, we set $\alpha = 0.001$ and $\lambda = 0.03$. The neural network is trained for 30 epochs with a learning rate of 0.01. Hyper-parameters including learning rate, α, λ, and size of the training set can be easily changed via the user interface as shown in Table 1.

4.2 Probabilistic Correctness of NeuReach

The following theorem shows that the error of the learned reachability function $R_{\hat{\theta}}$ can be bounded. Specifically, the difference between the error and the empirical loss is $O(\sqrt{\frac{1}{N}})$, where N is the size of the training set.

Theorem 1. *For any $\epsilon > 0$, and a random training set S with N i.i.d. samples, with probability at least $1 - 2\exp(-2N\epsilon^2)$, the following inequality holds,*

$$\mathbb{E}_X\left[\mathbb{I}\left(g_{\hat{\theta}}(X) - 1 > 0\right)\right] \leq \frac{1}{N}\sum_{i-1}^{N}\tilde{\ell}(g_{\hat{\theta}}(X_i) - 1) + \frac{12}{\alpha}L_g\sqrt{\frac{p}{N}} + \epsilon, \tag{5}$$

where p is the number of parameters, i.e. $\theta \in \mathbb{R}^p$, and $\tilde{\ell}(\cdot) = \min\{1, \ell(\cdot)\}$ is the truncated hinge loss, and L_g is the Lipschitz constant of g_θ w.r.t. θ.

Theorem 1 shows that by controlling ϵ and N, the actual error $\mathbb{E}_X\left[\mathbb{I}\left(g_{\hat{\theta}}(X) - 1 > 0\right)\right]$ can be made arbitrarily close to the empirical loss $\frac{1}{N}\sum_{i-1}^{N}\tilde{\ell}(g_{\hat{\theta}}(X_i)-1)$, with arbitrarily high probability. The *empirical loss* on the training set S can be made very small in practice due to the high capacity of the neural network. Of course, there is no free lunch, in general. In order to drive the empirical loss to 0, we might have to increase the number of parameters, which in turn increases the term $\frac{12}{\alpha}L_g\sqrt{\frac{p}{N}}$. Furthermore, the hyper-parameter λ also affects the empirical loss. A smaller λ results in lower empirical loss but more conservative reachsets. Actually, conservatism and accuracy are conflicting requirements. As shown in [21], when using reachability to verify safety, accuracy determines the soundness of the verification, while conservatism influences the sample efficiency. We wanted to focus more on soundness than on efficiency. Thus, a theoretical guarantee is derived for accuracy but not for conservatism.

Proof. Starting from the left hand side and using the definition of hinge loss, we get $\mathbb{E}_X\left[\mathbb{I}\left(g_{\hat{\theta}}(X) - 1 > 0\right)\right] \leq \mathbb{E}_X\left[\tilde{\ell}(g_{\hat{\theta}}(X) - 1)\right]$. By adding and subtracting the empirical loss term, we get:

$$\begin{aligned}
&\mathbb{E}_X\left[\tilde{\ell}(g_{\hat{\theta}}(X) - 1)\right] - \frac{1}{N}\sum_{i=1}^{N}\tilde{\ell}(g_{\hat{\theta}}(X_i) - 1) + \frac{1}{N}\sum_{i=1}^{N}\tilde{\ell}(g_{\hat{\theta}}(X_i) - 1)\\
&\leq \sup_{\theta \in \mathcal{W}}\left(\mathbb{E}_X\left[\tilde{\ell}(g_\theta(X) - 1)\right] - \frac{1}{N}\sum_{i=1}^{N}\tilde{\ell}(g_\theta(X_i) - 1)\right) + \frac{1}{N}\sum_{i=1}^{N}\tilde{\ell}(g_{\hat{\theta}}(X_i) - 1),
\end{aligned}$$

where the inequality follows from the definition of supremum.

Let $\mathcal{V} = \sup_{\theta \in \mathcal{W}}\left(\mathbb{E}_X\left[\tilde{\ell}(g_\theta(X) - 1)\right] - \frac{1}{N}\sum_{i=1}^{N}\tilde{\ell}(g_\theta(X_i) - 1)\right)$, i.e. the worst-case difference between the empirical average and the expectation of the loss. Note that $\mathcal{V}$ is a random quantity since $S = \{X_i\}_{i=1}^{N}$ is random. Next, we derive an upper bound on $\mathcal{V}$ that holds with high probability.

First, we derive an upper bound on $\mathbb{E}_S[\mathcal{V}]$. Let $\mathcal{G}$ be the function class containing g_θ parameterized by θ, i.e. $\mathcal{G} := \{g_\theta(\cdot) \mid \theta \in \mathcal{W}\}$. Similarly, $\mathcal{F} := \{\tilde{\ell}(g_\theta(\cdot) -$

$1) \,|\, \theta \in \mathcal{W}\}$. Applying $\mathcal{G}$ to the set of inputs S generates a new set $\mathcal{G}(S) := \{(g(X_1), g(X_2), \cdots, g(X_N)) : g \in \mathcal{G}\}$. Define $\mathcal{F}(S) := \{(f(X_1), \cdots, f(X_N)) : f \in \mathcal{F}\}$ in the same way.

Notice that $\mathcal{V}$ is the worst-case (among all $f_\theta \in \mathcal{F}$) gap between the expectation and the empirical average of $f_\theta(X)$. A fundamental result in PAC learning (Theorem 3.3 in [35]) shows that this gap can be bounded as $\mathbb{E}_S[\mathcal{V}] \leq 2\mathbb{E}_S[\mathtt{Rad}(\mathcal{F}(S))]$, where $\mathtt{Rad}(\mathcal{F}(S))$ is the Rademacher complexity [35] of $\mathcal{F}(S)$. Furthermore, notice that $\mathcal{F}(S)$ can be generated from $\mathcal{G}(S)$ by shifting it and composing it with $\tilde{\ell}$. It follows from Talagrand's contraction lemma [31] that $\mathbb{E}_S[\mathtt{Rad}(\mathcal{F}(S))] \leq 2L_{\tilde{\ell}}\mathbb{E}_S[\mathtt{Rad}(\mathcal{G}(S))]$, where $L_{\tilde{\ell}} = \frac{1}{\alpha}$ is the Lipschitz constant.

Finally, following from a conclusion on Rademacher complexity of Lipschitz parameterized function classes (See page 13 in [8]), we get $\mathbb{E}_S[\mathtt{Rad}(\mathcal{G}(S))] \leq 3L_g\sqrt{\frac{p}{N}}$. Therefore, we get

$$\mathbb{E}_S[\mathcal{V}] \leq \frac{12}{\alpha} L_g \sqrt{\frac{p}{N}}. \tag{6}$$

Then, applying McDiarmid's inequality [35] gives a high-probability bound on $\mathcal{V}$. That is,

$$\Pr_S\left(\left|\mathcal{V} - \mathbb{E}[\mathcal{V}]\right| \geq \epsilon\right) \leq 2\exp(-2N\epsilon^2).$$

Together with Eq. (6), we have $\mathcal{V} \leq \mathbb{E}[\mathcal{V}] + \epsilon \leq \frac{12}{\alpha}L_g\sqrt{\frac{p}{N}} + \epsilon$ with probability at least $1 - 2\exp(-2N\epsilon^2)$. This implies

$$\mathbb{E}_X\left[\mathbb{I}\left(g_{\hat{\theta}}(X) - 1 > 0\right)\right] \leq \frac{1}{N}\sum_{i-1}^{N} \tilde{\ell}(g_{\hat{\theta}}(X_i) - 1) + \frac{12}{\alpha} L_g \sqrt{\frac{p}{N}} + \epsilon,$$

with probability at least $1 - 2\exp(-2N\epsilon^2)$, which completes the proof. □

5 Experimental evaluation

We evaluated NeuReach on several benchmark systems including the Van der Pol oscillator, the Moore-Greitzer model of a jet engine, an 8-dimensional quadrotor controlled by a neural network [40], and an F-16 Ground Collision Avoidance system [28]. We also compare our method with DryVR [21]. Since NeuReach is fully data-driven and does not rely on the analytical model of the system, it would not make sense to compare against model-based methods like Hamilton-Jacobi reachability analysis [6], Flow* [11], C2E2 [18], or SReach [41]. Some of our benchmarks cannot be handled by these tools. Also, once the reachability function is learned, many reachsets can be computed very quickly by our method. Given that other tools need to compute the reachset from scratch for each new query, comparisons based on running times, would not make sense either.

5.1 Benchmark systems

The simulators available for the benchmark systems allow us to specify fixed time-steps Δt and a time bound T. As for the distribution P_2, we adopt the uniform distribution, i.e. $P_2 = \mathsf{Unif}(\{\Delta t, 2\Delta t, \cdots, \lfloor \frac{T}{\Delta t} \rfloor \Delta t\})$ (Recall, the definition of this distribution in Section 3). For a given initial set $\mathcal{X}_0$, $\mathcal{D}(\mathcal{X}_0)$ is defined as the uniform distribution on the boundary of $\mathcal{X}_0$. As shown in Corollary 1 of [43], the boundary of the reachable set of an initial set is equal to the reachable set of the initial set's boundary for ODEs. That is, if the estimated reachable set contains the reachable set of the initial set's boundary, it automatically contains that of the interior. Thus, we only sample points on the boundary of $\mathcal{X}_0$ to improve sample efficiency. As for the distribution P_1, we will give details for each benchmark below.

Van der Pol oscillator is a widely used 2-dimensional nonlinear model. An initial set $\mathcal{X}_0$ is a ball centered at c with radius r. The distribution P_1 for choosing $\mathcal{X}_0$ is specified by the distributions for choosing these parameters. In our experiments, we use $c \sim \mathsf{Unif}([1, 2] \times [2, 3])$ and $r \sim \mathsf{Unif}([0, 0.5])$. The time bound is set to $T = 4$, and time step is $\Delta t = 0.05$.

JetEngine model from [4] is also 2-dimensional and commonly used as a verification benchmark. Again, we use balls for the initial sets with $c \sim \mathsf{Unif}([0.3, 1.3] \times [0.3, 1.3])$ and $r \sim \mathsf{Unif}([0, 0.5])$. The time bound is set to $T = 10$, and time step is $\Delta t = 0.05$.

F-16 Ground Collision Avoidance System [**28**] is a challenging benchmark for formal analysis tools. This system consists of 16 state variables (See Table 1 in [28]) among which `Vt` and `alt` are air speed and altitude. The key safety property of interest is ground collision avoidance, and therefore, in our experiments we focus on estimating the reachset only for `Vt` and `alt`. We consider initial uncertainty in up to 6 state variables, $[\mathtt{Vt}, \alpha, \phi, \psi, \mathtt{Q}, \mathtt{alt}]$. The function `simulate(x0)` is designed to return projections of trajectories to `Vt` and `alt`, while `sample_X0()` returns 6-dimensional initial sets. We restrict the initial set to be hyper-rectangles as in [28]. An initial set $\mathcal{X}_0$ is determined by a center $c \in \mathbb{R}^6$ and a radius $r \in \mathbb{R}^6$ with $\mathcal{X}_0 = \{x \in \mathbb{R}^6 : c - r \leq x \leq c + r\}$. As for the distribution, we choose $c \sim \mathsf{Unif}([560, 600] \times [-0.1, 0.1] \times [0, \frac{\pi}{4}] \times [-\frac{\pi}{4}, \frac{\pi}{4}] \times [-0.1, 0.1] \times [70, 80])$ and $r \sim \mathsf{Unif}([0, 10] \times [0, 0.1] \times [0, \frac{\pi}{16}] \times [0, \frac{\pi}{8}] \times [0, 0.1] \times [0, 1])$. The time bound is set to $T = 20$, and time step is $\Delta t = \frac{1}{30}$. However, DryVR does not support hyper-rectangles as initial sets. Thus, we also use another setting for comparison where the initial sets are balls. To do this, we sample balls from a cube with $c \sim \mathsf{Unif}([-1, 1] \times \cdots \times [-1, 1])$ and $r \sim \mathsf{Unif}([0, 0.5])$. Then, we transform this ball to the original coordinate system by scaling each dimension. This setting is shown in Fig. 2 (Left) as F-16 (Spherical).

Quadrotor controlled by a neural controller is based on [40]. The state of the quadrotor system is $x = [p_x, p_y, p_z, v_x, v_y, v_z, \theta_x, \theta_y]$, and the control input

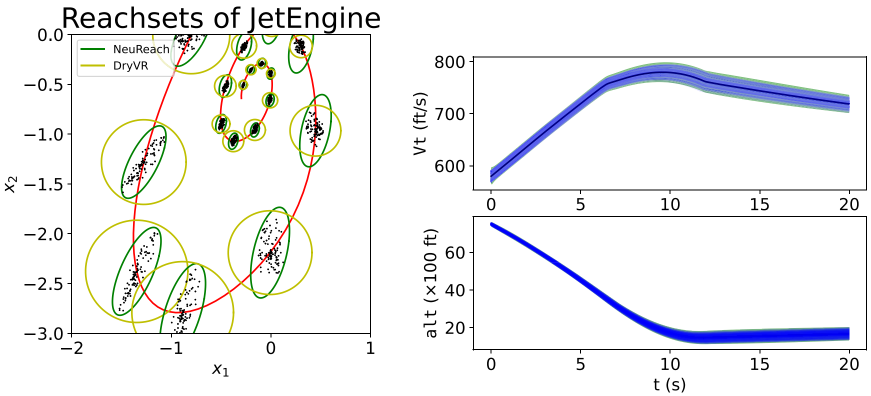


Fig. 1: Left: Some reachsets of JetEngine. Red curve is $\xi(\mathbb{E}\left[\mathcal{D}(\mathcal{X}_0)\right], \cdot)$. We randomly sample 100 trajectories starting from $\mathcal{X}_0$. Points on sampled trajectories are shown as black dots. Boundaries of the estimated reachsets at some selected time instants are shown. Clearly, ellipsoids can approximate the actual reachsets better; **Right**: A sample reachtube of F-16. Green region is the reachtube estimated by NeuReach, which is the union of all reachsets. Blue curves are sampled trajectories from the initial set. The blue region can be viewed as the actual reachtube. The estimated reachtube verifies the safety, i.e. `alt` > 0 always holds.

is $u := [a_z, \omega_x, \omega_y]$. We are only interested in estimating the reachability of the position variables, i.e., the first 3 dimensions of the state vector. We use balls for the initial sets with $c \sim \mathsf{Unif}([-1,1] \times \cdots \times [-1,1])$ and $r \sim \mathsf{Unif}([0, \sqrt{8}])$. The time bound is set to $T = 10$, and time step is $\Delta t = 0.05$.

5.2 Experimental results

Evaluation metrics. In order to evaluate the learned reachability function, we randomly sample 10 initial sets for testing. For each initial set $\mathcal{X}_0$, we then sample 100 trajectories starting from it. For every sampled time instant on the sampled trajectories, we check whether the state is contained in the estimated reachset and compute the empirical error (i.e., the frequency that a sample is not in the estimated reachset). In order to evaluate the conservatism of the over-approximations, we also compare the size of the over-approximations. For each initial set $\mathcal{X}_0$, we compute the total volume of the over-approximations $R(\mathcal{X}_0, t_i)$ where $t_i = \Delta t, 2\Delta t, \cdots, \lfloor \frac{T}{\Delta t} \rfloor \Delta t$. Then, the total volume averaged over 10 sampled initial sets are reported. Results are summarized in Figure 2 (Left). Please note that we use the default settings in Table 1 for all benchmarks.

All experiments were conducted on a Linux workstation with two Xeon Silver 4110 CPUs and 32 GB RAM. As shown in Figure 2 (Left), NeuReach learns an accurate reachability function for each benchmark. Please note that due to the complicated dynamics and the neural controller, the F-16 model and the quadrotor are beyond the reach of current model-based tools. As shown in Figure 1 (Right), NeuReach successfully verified the safety of the F-16 model.

Benchmark	NeuReach		DryVR	
	Volume	Error	Volume	Error
JetEngine	17.9	0.001	38.3	0.003
VanDerPol	39.2	0.001	76.4	0.002
Quadrotor	373.9	0.019	1025146.2	0.021
F-16 (Spherical)	28153.7	0.004	62651.5	0.004
F-16	31465.9	0.025	-	-

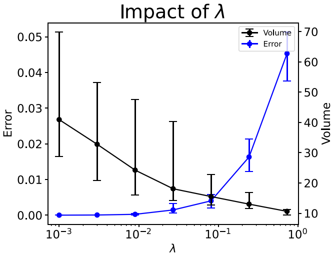


Fig. 2: Left: Volume and error of the estimated reachtube. Results are averaged over 10 random choices of $\mathcal{X}_0$; **Right**: Impact of λ. Error bars are the range over 10 runs.

Comparison with DryVR. DryVR [21] computes reachsets for spherical initial sets by learning a piece-wise exponential discrepancy (PED) function that bounds the sensitivity of the trajectories to the initial state. This function is of the form:

$$\beta(r,t) = rKe^{\sum_{j=1}^{i-1}\gamma_j(t_i-t_{i-1}+\gamma_i(t-t_{i-1})}, \forall t \in [t_{i-1}, t_i],$$

where r is the radius of the initial set, $[t_{i-1}, t_i]$ is the i-th time interval, and K, γ are learned parameters. For an spherical initial set $\mathcal{X}_0 = \mathcal{B}(c,r)$, the computed reachset is $R(\mathcal{B}(c,r),t) := \mathcal{B}(\xi(\mathbb{E}\left[\mathcal{D}(\mathcal{B}(c,r))\right], t), \beta(r,t))$, where $\mathcal{B}(c,r)$ is a ball centered at c with radius r. It is important to recall that, similar to other reachability tools, for every new initial set $\mathcal{X}_0$, DryVR computes the PED function and the reachset from scratch. For a fair comparison, we compute the parameters K and γ on the exact same training set as the one used in NeuReach and reuse the resulting PED for further queries.

Accuracy and conservatism. As shown in Figure 2 (Left), the reachsets estimated by NeuReach are tighter and more accurate than those computed by DryVR. There are two reasons for this. First, DryVR uses piece-wise exponential functions to capture the relationship between the initial radius and the radius at time t, while NeuReach uses more expressive neural networks. Second, the use of ellipsoids allows coordinate-specific accuracy. As seen in Figure 1, the reachset of JetEngine is not a perfect circle even if the initial set is a circle. Ellipsoids can approximate the actual reachsets better.

Running time. As expected, the training phase of NeuReach takes several minutes, but once a reachability function has been learned, computation of the reachset from a new initial set is very fast. For the quadrotor system, for example, this takes $\sim$ 0.3 ms on the aforementioned workstation. We believe that this makes NeuReach suitable for online safety checking and motion planning.

Impact of the hyper-parameter λ. λ influences the error and volume of the reachsets computed by NeuReach. Figure 2 (Right) shows the result of running NeuReach on JetEngine with different settings of λ. As expected, larger λ results in smaller reachsets but hurts the accuracy. On the other hand, we do not need to tune λ case by case. Note that we use $\lambda = 0.03$ for all the results in Figure 2 (Left), and it works reasonably well for all our benchmarks.

6 Conclusion

In this paper, we presented a tool for computing reachability of systems using machine learning. NeuReach can learn accurate reachability functions for complex nonlinear systems, including some that are beyond existing methods. From a learned reachability function, arbitrary reachtubes can be computed in milliseconds. There are several limitations in the current implementation of NeuReach. First, the simulator is assumed to be deterministic—this can be too restrictive for autonomous systems with complex perception and vehicle models. We plan to extend the theory and implementation to support more general simulators. Secondly, the over-approximations are restricted to be represented as ellipsoids. Other representations will be supported in the future.

References

1. dReach. http://dreal.github.io/dReach/
2. Althoff, M., Grebenyuk, D.: Implementation of interval arithmetic in CORA 2016. In: Proc. of the 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems. pp. 91–105 (2016)
3. Althoff, M., Grebenyuk, D., Kochdumper, N.: Implementation of Taylor models in cora 2018. In: Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems (2018). https://doi.org/10.29007/zzc7
4. Aylward, E.M., Parrilo, P.A., Slotine, J.J.E.: Stability and robustness analysis of nonlinear systems via contraction metrics and sos programming. Automatica **44**(8), 2163–2170 (2008)
5. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control. pp. 173–178. ACM (2017)
6. Bansal, S., Chen, M., Herbert, S., Tomlin, C.J.: Hamilton-jacobi reachability: A brief overview and recent advances. In: 2017 IEEE 56th Annual Conference on Decision and Control (CDC). pp. 2242–2253. IEEE (2017)
7. Bansal, S., Tomlin, C.: Deepreach: A deep learning approach to high-dimensional reachability. arXiv preprint arXiv:2011.02082 (2020)
8. Bartlett, P.: Lecture notes in theoretical statistics (February 2013), https://www.stat.berkeley.edu/~bartlett/courses/2013spring-stat210b/notes/14notes.pdf
9. Berndt, A., Alanwar, A., Johansson, K.H., Sandberg, H.: Data-driven set-based estimation using matrix zonotopes with set containment guarantees. arXiv preprint arXiv:2101.10784 (2021)
10. Bortolussi, L., Cairoli, F., Paoletti, N., Smolka, S.A., Stoller, S.D.: Neural predictive monitoring. In: International Conference on Runtime Verification. pp. 129–147. Springer (2019)
11. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: CAV. pp. 258–263. Springer (2013)

12. Cyranka, J., Islam, M.A., Byrne, G., Jones, P., Smolka, S.A., Grosu, R.: Lagrangian reachabililty. In: International Conference on Computer Aided Verification. pp. 379–400. Springer (2017)
13. Devonport, A., Arcak, M.: Data-driven reachable set computation using adaptive gaussian process classification and monte carlo methods. In: 2020 American Control Conference (ACC). pp. 2629–2634. IEEE (2020)
14. Devonport, A., Arcak, M.: Estimating reachable sets with scenario optimization. In: Learning for dynamics and control. pp. 75–84. PMLR (2020)
15. Devonport, A., Khaled, M., Arcak, M., Zamani, M.: PIRK: scalable interval reachability analysis for high-dimensional nonlinear systems. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 556–568. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_27, https://doi.org/10.1007/978-3-030-53288-8_27
16. Donzé, A., Jin, X., Deshmukh, J.V., Seshia, S.A.: Automotive systems requirement mining using breach. In: American Control Conference, ACC 2015, Chicago, IL, USA, July 1-3, 2015. p. 4097. IEEE (2015). https://doi.org/10.1109/ACC.2015.7171970, https://doi.org/10.1109/ACC.2015.7171970
17. Duggirala, P.S., Mitra, S., Viswanathan, M.: Verification of annotated models from executions. In: EMSOFT (2013)
18. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2e2: A verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 68–82. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
19. Everett, M., Habibi, G., Sun, C., How, J.P.: Reachability analysis of neural feedback loops. IEEE Access **9**, 163938–163953 (2021)
20. Fan, C., Kapinski, J., Jin, X., Mitra, S.: Locally optimal reach set over-approximation for nonlinear systems. In: EMSOFT. pp. 6:1–6:10. ACM (2016)
21. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: Data-driven verification and compositional reasoning for automotive systems. In: Computer Aided Verification. pp. 441–461. Springer International Publishing (2017)
22. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. pp. 531–538 (2016). https://doi.org/10.1007/978-3-319-41528-4_29, https://doi.org/10.1007/978-3-319-41528-4_29
23. Fan, D.D., Agha-mohammadi, A.a., Theodorou, E.A.: Deep learning tubes for tube mpc. arXiv preprint arXiv:2002.01587 (2020)
24. Fijalkow, N., Ouaknine, J., Pouly, A., Sousa-Pinto, J.a., Worrell, J.: On the decidability of reachability in linear time-invariant systems. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. p. 77–86. HSCC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3302504.3311796, https://doi.org/10.1145/3302504.3311796
25. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 6806, pp. 379–395. Springer (2011)

26. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: International Joint Conference on Automated Reasoning. pp. 286–300. Springer (2012)
27. Gurung, A., Ray, R., Bartocci, E., Bogomolov, S., Grosu, R.: Parallel reachability analysis of hybrid systems in xspeed. Int. J. Softw. Tools Technol. Transf. **21**(4), 401–423 (2019). https://doi.org/10.1007/s10009-018-0485-6, https://doi.org/10.1007/s10009-018-0485-6
28. Heidlauf, P., Collins, A., Bolender, M., Bak, S.: Verification challenges in f-16 ground collision avoidance and other automated maneuvers. In: ARCH@ ADHS. pp. 208–217 (2018)
29. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: ACM Symposium on Theory of Computing. pp. 373–382 (1995), citeseer.nj.nec.com/henzinger95whats.html
30. Jiang, F., Chou, G., Chen, M., Tomlin, C.J.: Using neural networks to compute approximate and guaranteed feasible hamilton-jacobi-bellman pde solutions. arXiv preprint arXiv:1611.03158 (2016)
31. Ledoux, M., Talagrand, M.: Probability in Banach Spaces: isoperimetry and processes. Springer Science & Business Media (2013)
32. Lew, T., Pavone, M.: Sampling-based reachability analysis: A random set theory approach with adversarial sampling. arXiv preprint arXiv:2008.10180 (2020)
33. Maidens, J., Arcak, M.: Reachability analysis of nonlinear systems using matrix measures. Automatic Control, IEEE Transactions on **60**(1), 265–270 (2015)
34. Mitra, S.: Verifying Cyber-Physical Systems: A Path to Safe Autonomy. MIT Press (2021), https://mitpress.mit.edu/contributors/sayan-mitra
35. Mohri, M., Rostamizadeh, A., Talwalkar, A.: Foundations of machine learning. MIT press (2018)
36. Niarchos, K., Lygeros, J.: A neural approximation to continuous time reachability computations. In: Proceedings of the 45th IEEE Conference on Decision and Control. pp. 6313–6318. IEEE (2006)
37. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
38. Phan, D., Paoletti, N., Zhang, T., Grosu, R., Smolka, S.A., Stoller, S.D.: Neural state classification for hybrid systems. In: International Symposium on Automated Technology for Verification and Analysis. pp. 422–440. Springer (2018)
39. Shmarov, F., Zuliani, P.: Probreach: verified probabilistic delta-reachability for stochastic hybrid systems. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 134–139 (2015)
40. Sun, D., Jha, S., Fan, C.: Learning certified control using contraction metric. arXiv preprint arXiv:2011.12569 (2020)
41. Wang, Q., Zuliani, P., Kong, S., Gao, S., Clarke, E.M.: Sreach: A bounded model checker for stochastic hybrid systems. arXiv preprint arXiv:1404.7206 (2014)
42. Willems, J.C.: The behavioral approach to open and interconnected systems. IEEE Control Systems Magazine **27**(6), 46–99 (2007). https://doi.org/10.1109/MCS.2007.906923

43. Xue, B., Easwaran, A., Cho, N.J., Fränzle, M.: Reach-avoid verification for nonlinear systems based on boundary analysis. IEEE Transactions on Automatic Control **62**(7), 3518–3523 (2016)
44. Xue, B., Zhang, M., Easwaran, A., Li, Q.: PAC model checking of black-box continuous-time dynamical systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(11), 3944–3955 (2020)

Inferring Invariants with Quantifier Alternations: Taming the Search Space Explosion*

Jason R. Koenig[1] (✉), Oded Padon[2], Sharon Shoham[3], and Alex Aiken[1]

[1] Stanford University, Stanford, CA, USA {jrkoenig,aaiken}@stanford.edu
[2] VMware Research, Palo Alto, CA, USA oded.padon@gmail.com
[3] Tel Aviv University, Tel Aviv, Israel sharon.shoham@gmail.com

Abstract. We present a PDR/IC3 algorithm for finding inductive invariants with quantifier alternations. We tackle scalability issues that arise due to the large search space of quantified invariants by combining a breadth-first search strategy and a new syntactic form for quantifier-free bodies. The breadth-first strategy prevents inductive generalization from getting stuck in regions of the search space that are expensive to search and focuses instead on lemmas that are easy to discover. The new syntactic form is well-suited to lemmas with quantifier alternations by allowing both limited conjunction and disjunction in the quantifier-free body, while carefully controlling the size of the search space. Combining the breadth-first strategy with the new syntactic form results in useful inductive bias by prioritizing lemmas according to: (i) well-defined syntactic metrics for simple quantifier structures and quantifier-free bodies, and (ii) the empirically useful heuristic of preferring lemmas that are fast to discover. On a benchmark suite of primarily distributed protocols and complex Paxos variants, we demonstrate that our algorithm can solve more of the most complicated examples than state-of-the-art techniques.

Keywords: invariant inference · quantifier alternation · PDR/IC3

1 Introduction

Invariant inference is a long-standing problem in formal methods, due to the desire for verified systems without the cost of manually writing invariants. For complex unbounded systems the required invariants often involve quantifiers, including quantifier alternations. For example, an invariant for a distributed system may need to quantify over an unbounded number of nodes, messages, etc. Furthermore, it may need to nest quantifiers in alternation (between ∀ and ∃) to capture the system's correctness arguments. For example, one crucial invariant of the Paxos consensus protocol [22] is "every decision must come from a quorum of votes", i.e. $\forall decision. \exists quorum. \forall node.\ node \in quorum \Rightarrow node\ voted\ for\ decision.$

* This research received funding from the European Union's Horizon 2020 research and innovation programme grant No. [759102-SVIS], BSF grant No. 2016260, ISF grant No. 1810/18, and NSF grants CCF-1160904 and CCF-1409813.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 338–356, 2022.
https://doi.org/10.1007/978-3-030-99524-9_18

We show that automatically inferring such invariants is possible for systems beyond the current state of the art by addressing several scalability issues that arise as the complexity of systems and their invariants increases.

Many recent successful invariant inference techniques, including ours, are based on PDR/IC3 [3,5]. PDR/IC3 is an algorithmic framework for finding inductive invariants *incrementally*, rather than attempting to find the entire inductive invariant at once. PDR/IC3 progresses by building a collection of *lemmas*, organized into *frames* labeled by number of steps from the initial states, until eventually some of these lemmas form an inductive invariant. New lemmas are generated by *inductive generalization*, where a given (often backward reachable) state is generalized to a formula that excludes it and is inductive *relative* to a previous frame. Inductive generalization therefore plays a key role in PDR/IC3 implementations. Specifically, extending PDR/IC3 to a new domain of lemmas requires a suitable inductive generalization procedure.

Techniques for inductive generalization, and more broadly for generating formulas for inductive invariants, are varied, including interpolation [25], quantifier elimination [20], model-based techniques [18], and syntax guided synthesis [6,31]. Almost all of these existing techniques target either quantifier-free or universally quantified invariants. While it is sometimes possible to manually transform a transition system to eliminate some of the need for quantifiers [8], doing so is difficult and requires some knowledge of the fully quantified invariant.

We present a system that can infer quantified invariants with alternations based on *quantified separation*, which was introduced in [19]. Roughly, a separation query asks whether there is a quantified formula, a *separator*, that evaluates to true on a given set of models and to false on another given set of models. While [19] used separation (as a black box) to implement inductive generalization and described the first PDR/IC3 implementation that finds invariants with quantifier alternations, it did not scale to challenging protocols such as Paxos and its variants. These protocols require invariants with many symbols and quantifiers, and the search space for quantified separators explodes as the number of symbols in the vocabulary and number of quantifiers increases. In contrast, this work presents a technique that can automatically find such complex invariants.

When targeting complex invariants, there are two main challenges for inductive generalization: (i) the run time of each individual query; and (ii) overfitting, i.e., learning a lemma that eliminates the given state but does not advance the search for an inductive invariant. We tackle both problems via two strategies: the first integrates inductive generalization with separation in a breadth-first way, and the second defines a new form, k-term pDNF, for the quantifier-free Boolean structure of the separators.

Integrating quantified separation with inductive generalization enables us to effectively use a breadth-first rather than a depth-first search strategy for the quantifiers of potential separators: we search in multiple parts of the search space simultaneously rather than exhaustively exploring one region before moving to the next. Beyond enabling parallelism, and thus faster wall-clock times, this restructuring can change which solution is found by allowing easy-to-search regions to find a solution first. We find that these easier-to-find formulas generalize better (i.e., avoid overfitting).

Using k-term pDNF narrows the search space for lemmas with quantifier alternations. Universally quantified invariants can be split into universally quantified clauses by transformation into conjunctive normal form (CNF). Accordingly, most PDR/IC3 based techniques find invariants as conjunctions of possibly quantified clauses. However, invariants with quantifier alternations may require conjunction inside quantified lemmas (e.g., consider $\forall x. \exists y. p(y) \wedge r(x, y)$). Using multiple clauses per lemma (k-clause CNF) creates a significantly larger search space, impeding scalability. Using disjunctive normal form (DNF) suffers from the same problem. We introduce k-term pDNF, a class of Boolean formulas inspired by human-written invariants that allows both limited conjunction and disjunction while keeping the search space manageable. Many of the lemmas arising in our evaluation that require many clauses in CNF are only 2-term pDNF. We modify separation to search for lemmas of this form, leading to a reduced search space compared to CNF or DNF, resulting in both faster inductive generalization and less overfitting.

We evaluate our technique on a benchmark suite that includes challenging distributed protocols. Inferring invariants with quantifier alternations has recently drawn significant attention, with recent works, [19,11], presenting techniques based on PDR/IC3 that find invariants with quantifier alternations but do not scale to complex protocols such as Paxos. Very recently, [14] and [12] presented enumeration-based and PDR/IC3-based techniques, respectively, which find the invariant for simple variants of Paxos, but do not scale to more complex variants. Our experiments show that our separation-based approach significantly advances the state-of-the-art, and scales to several Paxos variants which are unsolved by prior works. We also present an ablation study that investigates the individual effect of key features of our technique.

This work makes the following contributions:

1. An algorithm for inductive generalization in PDR/IC3 (Section 3) based on quantified separation that explores the search space in a parallel, breadth-first way and thus focuses on lemmas that are easy to discover without requiring *a priori* knowledge of the search space.
2. A syntactic form of lemmas (k-pDNF, Section 4) that is well-suited for invariants with quantifier alternations.
3. A combined system (Section 5) able to infer the invariants of challenging protocols with quantifier alternations, including complex Paxos variants.
4. A comprehensive evaluation (Section 6) on a large benchmark suite including complex Paxos variants, comparisons with a variety of state-of-the-art tools, and an ablation study exploring the effects of key features of our technique.

2 Background

We review first-order logic, quantified separation, the invariant inference problem, and PDR/IC3.

First-Order Logic. We consider formulas in many-sorted first-order logic with uninterpreted functions and equality. A *signature* consists of a finite set of sorts and sorted constant, relation, and function symbols. A first-order *structure* over a given signature consists of a *universe* set of sorted elements along with interpretations for each symbol. A structure is finite when its universe is finite. We use the standard definitions for *term*, *atomic formula*, *literal*, *quantifier-free formula*. *Quantified* formulas may contain universal ($\forall$) and existential ($\exists$) quantifiers with sorted variables (e.g. $\forall x{:}s_1.\, p$). A formula is in *prenex normal form* if it consists of a (possibly empty) quantification *prefix* followed by a quantifier-free *matrix*. Any formula can be mechanically transformed into an equivalent prenex formula. A structure M satisfies a formula p, written $M \models p$, if the formula is true when the symbols in p are interpreted according to M under the usual semantics. If such an M exists, then p is *satisfiable* and M is a *model* of p.

Quantified Separation. To generate candidate lemmas, we use *quantified separation* [19]. Given a set of *structure constraints* and a predetermined space of formulas, separation produces a separator formula p from the space that satisfies the constraints, or reports UNSEP if no such p exists. The constraints are either *positive* (a structure M where $M \models p$), *negative* (a structure M where $M \not\models p$) or *implication* (a pair of structures M, M' where $M \models p \Rightarrow M' \models p$). Separation producing prenex formulas under some assumptions (satisfied by practical examples) is NP-complete [19], and can be solved by translation to SAT.

Invariant Inference. The invariant inference problem is to compute an *inductive invariant* for a given *transition system*, which shows that only *safe* states are reachable from the *initial* states. We consider a transition system to be a set of *states* as structures over some signature satisfying an axiom Ax, some initial states satisfying $Init$, a transition formula Tr which can contain primed symbols (x') representing the post-state, and safe states satisfying $Safe$. We define *bad* states as $\neg Safe$. We define single-state implication, written $A \Rightarrow B$, as $\textsc{Unsat}(A \wedge Ax \wedge \neg B)$ and two-state implication across transitions, written $A \Rightarrow \mathrm{wp}(B)$,[4] as $\textsc{Unsat}(A \wedge Ax \wedge Tr \wedge Ax' \wedge \neg B')$. An *inductive invariant* is a formula I satisfying:

$$Init \Rightarrow I \qquad (1) \qquad\qquad I \Rightarrow \mathrm{wp}(I) \qquad (2) \qquad\qquad I \Rightarrow Safe \qquad (3)$$

Together, (1) and (2) mean that I is satisfied by all reachable states, and (3) ensures the system is safe. We only consider invariant inference for safe systems.

PDR/IC3. PDR/IC3 is an invariant inference algorithm first developed for finite state model checking [3] and later extended to various classes of infinite-state systems. We describe PDR/IC3 as in [17]. PDR/IC3 maintains *frames* F_i as conjunctions of formulas (*lemmas*) representing overapproximations of the states reachable in at most i transitions from $Init$. Finite frames ($i = 0, \ldots, n$) and the frame at infinity ($i = \infty$) satisfy:

[4] Our use of wp is inspired by predicate transformers, but we define it via satisfiability.

$$Init \Rightarrow F_0 \quad (4) \qquad F_i \Rightarrow F_{i+1} \quad (5) \qquad F_n \Rightarrow F_\infty \quad (6)$$

$$F_i \Rightarrow \text{wp}(F_{i+1}) \quad (7) \qquad F_\infty \Rightarrow \text{wp}(F_\infty) \quad (8)$$

Conditions (4), (5), and (6) mean $Init \Rightarrow F_i$ for all i, and we ensure this by restricting frames to subsets of the prior frame, when taken as sets of lemmas. Conditions (7) and (8) say each frame is *relatively inductive* to the prior frame, except F_∞ which is relatively inductive to itself and thus inductive for the system. To initialize, the algorithm adds the (conjuncts of) *Init* and *Safe* as lemmas to F_0. The algorithm then proceeds by adding lemmas to frames using either *pushing* or *inductive generalization* while respecting this meta-invariant, gradually tightening the bounds on reachability until $F_\infty \Rightarrow Safe$. We can push a lemma $p \in F_i$ to F_{i+1}, provided $F_i \Rightarrow \text{wp}(p)$. When a formula is pushed, the stronger F_{i+1} may permit us to push one or more other formulas, possibly recursively, and so we always push until a fixpoint is reached. Any mutually relatively inductive set of lemmas do not have a finite fixpoint, and we detect these sets (by checking for $F_i = F_{i+1}$) and move them to F_∞.

If the algorithm cannot push a lemma p_a beyond frame i, there is a model of $\neg(F_i \Rightarrow \text{wp}(p_a))$, which is a transition $s \to t$ where $s \in F_i$ and $t \not\models p_a$. We call the pre-state s a *pushing preventer* of p_a. To generate new lemmas, we *block* the pushing preventer s in F_i by first recursively blocking all predecessors of s that are still in F_{i-1}, and then using an inductive generalization (IG) query to *learn* a new lemma that eliminates s. An IG query finds a formula p satisfying:

$$s \not\models p \quad (9) \qquad Init \Rightarrow p \quad (10) \qquad F_{i-1} \wedge p \Rightarrow \text{wp}(p) \quad (11)$$

If we can learn such a lemma, it can be added to F_i and all previous frames, and removes at least the state s stopping p_a from being pushed. Classic PDR/IC3 always chooses to block the pushing preventer of a safety property (lemma from *Safe*) or a predecessor thereof, but other strategies have been considered [17]. The technique used to solve IG queries controls what kind of invariants we are able to discover. In this work we use separation to solve for p, which lets us infer invariants with quantifier alternations.

3 Breadth-First Inductive Generalization with Separation

Inductive generalization is the core of PDR/IC3, and improving it comes in two flavors: making individual queries faster, and generating better lemmas that are more general. We address both of these concerns by restructuring the search to be *breadth-first* rather than *depth-first*. We first discuss naively solving an IG query with separation (as in [19]), then present an algorithm that restructures the search in a breadth-first manner.

3.1 Naive Inductive Generalization with Separation

An IG query is solved in [19] with separation by a simple refinement loop, which performs a series of separation queries with an incrementally growing set of structure constraints. Starting with a negative constraint s for the state to block, we ask for a separator p and check if eqs. (10) and (11) hold for p using a standard SMT solver. If both hold, p is a solution to the IG query. Otherwise, the SMT solver produces a model which becomes either a positive constraint (corresponding to an initial state p violates) or an implication constraint (a transition edge that shows p is not relatively inductive to F_{i-1}), respectively.

At a high level, the SAT-based algorithm for separation from [19] uses Boolean variables to encode the kind ($\forall/\exists$) and sort of each quantifier, and additional variables for the presence of each syntactically valid literal in each clause in the matrix, which is in CNF. It then translates each structure constraint into a Boolean formula over these variables such that satisfying assignments encode formulas with the correct truth value for each structure. The details of the translation to SAT are not relevant here, except a few key points: (i) separation considers each potential quantifier prefix essentially independently, (ii) complex IG queries can result in hundreds or thousands of constraints, and (iii) prefixes, as partitions of the space of possible separators, vary greatly in how quickly they can be explored. Further, with the black box approach where the prefixes are considered internally by the separation algorithm, even if the separation algorithm uses internal parallelism as suggested in [19], there is still a serialization step when a new constraint is required. As a consequence of (ii) and (iii), a significant failure mode of this naive approach is that the search becomes stuck generating more and more constraints for difficult parts of the search space that ultimately do not contain an easy-to-discover solution to the IG query.

3.2 Prefix Search at the Inductive Generalization Level

To fix the problems with the naive approach, we propose lifting the choice of prefix to the IG level, partitioning a single large separation query into a query for each prefix. Each sub-query can be explored in parallel, and each can proceed independently by querying for new constraints (using eqs. (10) and (11) as before) without serializing by waiting for other prefixes. We call this a *breadth-first* search, because the algorithm can spend approximately equal time on many parts of the search space, instead of a *depth-first* search which exhausts all possibilities in one region before moving on to the next. When regions have greatly varying times to search, the breadth-first approach prevents expensive regions from blocking the search in cheaper regions. This improvement relies on changing the division between separation and inductive generalization: without the knowledge of the formulas (eqs. (10) and (11)) that generate constraints, the separation algorithm cannot generate new constraints on its own.

A complicating factor is that in addition to prefixes varying in difficulty, sometimes there are entire classes of prefixes that are difficult. For example, many IG queries have desirable universal-only solutions, but spend a long time searching for separators with alternations, as there are far more distinct prefixes with alternations than those with only universals. To address this problem, we define possibly overlapping sets of prefixes, called *prefix categories*, and ensure the algorithm spends approximately equal time searching for solutions in

```
def IG($s$: state, $i$: frame):
    $\forall P.\ C(P) = \{\text{Negative}(s)\}$;
    for $i = 1 \ldots N$ in parallel:
        while true:
            $P$ = next-prefix();
            while true:
                $p$ = separate $C(P)$;
                if $p$ is UNSEP:
                    break
                elif any $c \in R_C(P)$ and $p \not\models c$:
                    add $c$ to $C(P)$
                elif ($c$ := SMT check eqs. (10) and (11)) $\neq$ UNSAT:
                    add $c$ to $C(P)$
                else:
                    return $p$ as solution
```

Fig. 1. Pseudocode for our proposed inductive generalization algorithm.

each category (e.g., universally quantified invariants, invariants with at most one alternation and at most one repeated sort). Within each category, we order prefixes to further bias towards likely solutions: first by smallest quantifier depth, then fewest alternations, then those that start with a universal, and finally by smallest number of existentials.

3.3 Algorithm for Inductive Generalization

We present our algorithm for IG using separation in Figure 1. Our algorithm has a fixed number N of worker threads which take prefixes from a queue subject to prefix restrictions, and perform a separation query with that prefix. Each worker thread calls next-prefix() to obtain the next prefix to consider, according to the order discussed in the previous section. To solve a prefix P, a worker performs a refinement loop as in the naive algorithm, building a set of constraints $C(P)$ until a solution to the IG query is discovered or separation reports UNSEP.

While we take steps to make SMT queries for new constraints as fast as possible (Section 5.4), these queries are still expensive and we thus want to re-use constraints between prefixes where it is beneficial. Re-using every constraint discovered so far is not a good strategy as the cost of checking upwards of hundreds of constraints for every candidate separator is not justified by how frequently they actually constrain the search. Instead, we track a set of *related constraints* for a prefix P, $R_C(P)$. We define related constraints in terms of *immediate sub-prefixes* of P, written $S(P)$, which are prefixes obtained by dropping exactly one quantifier from P, i.e. the quantifiers of $P' \in S(P)$ are a subsequence of those in P with one missing. We then define $R_C(P) = \cup_{P' \in S(P)} C(P')$, i.e. the related constraints of P are all those used by immediate sub-prefixes. While $S(P)$ considers only immediate sub-prefixes, constraints may propagate from non-immediate sub-prefixes as the algorithm progresses.

Constraints from sub-prefixes are used because the possible separators for those queries are also possible separators for the larger prefix. Thus the set of constraints from sub-prefixes will definitely eliminate some potential separators, and in the usual case where the sub-prefixes have converged to UNSEP, will rule out an entire section of the search space. We also opportunistically make use of known constraints for the same prefix generated in prior IG queries, as long as those constraints still satisfy the current frame.

Overall, the algorithm in Figure 1 uses parallelism across prefixes to generate independent separation queries in a breadth-first way, while carefully sharing only useful constraints. From the perspective of the global search for an inductive invariant the algorithm introduces two forms of inductive bias: (i) explicit bias arising from controlling the order and form of prefixes (Section 3.2), and (ii) implicit bias towards formulas which are easy to discover.

4 *k*-Term Pseudo-DNF

We now consider the search space for quantifier-free matrices, and introduce a syntactic form that shrinks the search space while still allowing common invariants with quantifier alternations to be expressed.

Conjunctive and disjunctive normal forms (CNF and DNF) are formulas that consist of a conjunction of *clauses* (CNF) or a disjunction of *cubes* (DNF), where clauses and cubes are disjunctions and conjunctions of literals, respectively: For example, $(a \vee b \vee \neg c) \wedge (b \vee c)$ is in CNF and $(a \wedge \neg c) \vee (\neg a \wedge b)$ is in DNF. We further define k-clause CNF and k-term DNF as formulas with at most k clauses and cubes, respectively.

In [19] separation is performed by finding a matrix in k-clause CNF, biasing the search by minimizing the sum of the number of quantifiers and k. We find that both CNF and DNF are not good fits for the formulas in human-written invariants. For example, consider the following formula from Paxos:

$$\begin{aligned}&\forall r_1, r_2, v_1, v_2, q. \exists n. r_1 < r_2 \wedge \text{proposal}(r_2, v_2) \wedge v_1 \neq v_2 \\ &\rightarrow \text{member}(n, q) \wedge \text{left-round}(n, r_1) \wedge \neg\text{vote}(n, r_1, v_1)\end{aligned}$$

To write this in CNF, we need to distribute the antecedent over the conjunction, obtaining the 3-clause formula:

$$\begin{aligned}&(r_1 < r_2 \wedge \text{proposal}(r_2, v_2) \wedge v_1 \neq v_2 \rightarrow \text{member}(n, q)) \wedge \\ &(r_1 < r_2 \wedge \text{proposal}(r_2, v_2) \wedge v_1 \neq v_2 \rightarrow \text{left-round}(n, r_1)) \wedge \\ &(r_1 < r_2 \wedge \text{proposal}(r_2, v_2) \wedge v_1 \neq v_2 \rightarrow \neg\text{vote}(n, r_1, v_1))\end{aligned}$$

When written without →, this matrix has the form $\neg a \vee \neg b \vee c \vee (d \wedge e \wedge \neg f)$, which is already in DNF. Under the k-term DNF, however, the formula requires a single-literal cube for each antecedent literal, i.e. $k = 4$. Because of the quantifier alternation, we cannot split this formula into cubes or clauses, and so a search over either CNF or DNF must consider a significantly larger search space.To solve these issues, we define a variant of DNF, k-term pseudo-DNF (k-pDNF), where one cube is negated, yielding as many individual literals as needed:

Definition 1 (k-term pseudo-DNF). *A quantifier-free formula φ is in k-term pseudo-DNF for $k \geq 1$ if $\varphi \equiv \neg c_1 \vee c_2 \vee \ldots \vee c_k$, where $c_1, \ldots, c_k$ are* *cubes. Equivalently, φ is in k-term pDNF if there exists $n \geq 0$ such that $\varphi \equiv \ell_1 \vee \ldots \vee \ell_n \vee c_2 \vee \ldots \vee c_k$, where $\ell_1, \ldots, \ell_n$ are literals and $c_2, \ldots, c_k$ are cubes.*

Note that 1-term pDNF is equivalent to 1-clause CNF, i.e. a single clause. 2-term pDNF correspond to formulas of the form (cube) → (cube). Such formulas are sufficient for all but a handful of the lemmas required for invariants in our benchmark suite. An exception is the following, which has one free literal and two cubes (so it is 3-term pDNF):

$$\begin{aligned} &\forall v_1.\, \exists n_1, n_2, n_3, v_2, v_3. \\ &\quad (d(v_1) \to \neg m(n_1) \wedge u(n_1, v_1)) \vee \\ &\quad (\neg m(n_2) \wedge \neg m(n_3) \wedge u(n_2, v_2) \wedge u(n_3, v_3) \wedge v_2 \neq v_3) \end{aligned}$$

For a fixed k, k-clause CNF, k-term DNF, and k-term pDNF all have the same-size search space, as the SAT query inside the separation algorithm will have one indicator variable for each possible literal in each clause or cube. The advantage of pDNF is that it can express more invariant lemmas with a small k, reducing the size of the search space while still being expressive. We can also see pDNF as a compromise between CNF and DNF, and we find that pDNF is a better fit to the matrices of invariants with quantifier alternation.

5 An Algorithm for Invariant Inference

We now take a step back to consider the high-level PDR/IC3 structure of our algorithm. We have described how our algorithm performs inductive generalization (Sections 3 and 4), which is the central ingredient. We next discuss blocking states that are not backward reachable from a bad state as a heuristic for finding additional useful lemmas. We then discuss how we can search for formulas in the EPR logic fragment and techniques to increase the robustness of SMT solvers. Finally, we give a complete description of our proposed algorithm.

5.1 May-proof-obligations

In classic PDR/IC3, the choice of pushing preventer to block is always that of a safety property. [17] proposed a heuristic that in our terminology is to block the pushing preventer of other existing lemmas, under the heuristic assumption that current lemmas in lower frames are part of the final invariant but lack a supporting lemma to make them inductive. The classic blocked states are known as *must-proof-obligations*, as they are states that must be eliminated somehow to prove the safety property. In contrast, these heuristic states are *may-proof-obligations*, as they may or may not be necessary to block. Our algorithm selects these lemmas at random, biased towards lemmas with smaller matrices.

To block a state, we first recursively block its predecessors in the prior frame, if they exist. For may-proof-obligations,[5] this recursion can potentially reach all

[5] For unsafe transition systems, this can also occur for must-proof-obligations.

the way to an initial state in F_0, and thus proves that the entire chain of states is reachable— i.e., the states cannot be blocked. This fact shows that the original lemma is not part of any final invariant and cannot be pushed past its current frame; it also provides a positive structure constraint useful for future IG queries.

5.2 Multi-block Generalization

After an IG query blocking state s is successful, the resulting lemma p may cause the original lemma that created s to be pushed to the next frame. If not, there will be a new pushing preventer s'. If s' is in the same frame, we can ask whether there is a single IG solution formula p_1 which blocks both s and s'. If we can find such a p_1, it is more likely to generalize past s and s', and we should prefer p_1. This is straightforward to do with separation: we incrementally add another negative constraint to the existing separation queries. To implement *multi-block generalization*, we continue an IG query if the new pushing preventer is suitable (i.e. exists and is in the same frame), accumulating as many negative constraints as we can until we do not have a suitable state or we have spent as much time as the original query. This timeout guarantees we do not spend more than half of our time on generalization, and protects us in the case that the new set of states cannot be blocked together with a simple formula.

5.3 Enforcing EPR

Effectively Propositional Reasoning (EPR, [28]) is a fragment of many-sorted first-order logic in which satisfiability is decidable and satisfiable formulas always have a finite model. The essence of EPR is to limit function symbols, both in the signature and from the Skolemization of existentials, to ensure only a finite number of ground terms can be formed. EPR ensures this property by requiring that there be no cycles in the directed graph with an edge from each domain sort to the codomain sort for every (signature and Skolem) function symbol. For example, $(\forall x{:}S.\, \varphi_1) \vee (\exists y{:}S.\, \varphi_2)$ is in EPR, but $\forall x{:}S.\, \exists y{:}S.\, \varphi_3$ is not in EPR as the Skolem function for y introduces an edge from sort S to itself. The acyclicity requirement means that EPR is not closed under conjunction, and so is best thought of as a property of a whole SMT query rather than of individual lemmas. Despite these restrictions, EPR can be used to verify complex distributed protocols [28].

For invariant inference with PDR/IC3, the most straightforward way to enforce acyclicity is to decide *a priori* which edges are allowed, and to not infer lemmas with disallowed Skolem edges. In practice, enforcing EPR means simply skipping prefixes during IG queries that would create disallowed edges. Without this fixed set of allowed edges, adding a lemma to a frame may prevent a necessary lemma from being added to the frame in a later iteration, as PDR/IC3 lacks a way to remove lemmas from frames. Requiring the set of allowed edges as input is a limitation of our technique and other state-of-the-art approaches (e.g. [14]). We hope that future work expands the scope of decidable logic fragments, so that systems require less effort to model in such a fragment. It is also possible that our algorithm could be wrapped in an outer search over the possible acyclic sets of edges.

Because separation produces prenex formulas, some EPR formulas would be disallowed without additional effort (e.g. a prenex form of $(\forall x{:}S.\, \varphi_1) \vee (\exists y{:}S.\, \varphi_2)$ is $\forall x{:}S.\, \exists y{:}S.\, (\varphi_1) \vee (\varphi_2))$. In our implementation, we added an option where separation produces prenex formulas that may not be in EPR directly, but where the scope of the quantifiers can be *pushed down* into the pDNF disjunction to obtain an EPR formula. Extra SAT variables are introduced that encode whether a particular quantified variable appears in a given disjunct, and we add the constraint that the quantifiers are nested consistently and in such a way as to be free of disallowed edges. Because this makes separation queries more difficult, we only enable this mode for the single example that requires non-prenex EPR formulas.

5.4 SMT Robustness

Even with EPR restrictions, some SMT queries we generate are difficult for the SMT solvers we use (Z3 [4] and CVC5[6]), sometimes taking minutes, hours, or longer. This wide variation of solving times is significant because separation, and thus IG queries, cannot make progress without a new structure constraint. We adopt several strategies to increase robustness: periodic restarts, running multiple instances of both solvers in parallel, and *incremental queries*. Our incremental queries send the formulas to the SMT solver one at a time, asserting a subset of the input. An UNSAT result from a subset can be returned immediately, and a SAT result can be returned if there is no un-asserted formula violated by the model. Otherwise, one of the violating formulas is asserted, and the process repeats. This process usually avoids asserting all the discovered lemmas from a frame, which significantly speeds up many of the most difficult queries, especially those with dozens of lemmas in a frame or those not in EPR.

5.5 Complete Algorithm

Figure 2 presents the pseudocode for our algorithm, which consists of two parallel tasks (learning and heuristic), each using half of the available parallelism to discharge IG queries, and pushing to fixpoint after adding any lemmas. In this listing, the to-block(ℓ) function computes the state and frame to perform an IG query in order to push ℓ (i.e. the pushing preventer of ℓ or a possibly multi-step predecessor thereof). The heuristic task additionally may find reachable states, and thus mark lemmas as bad. We cancel an IG query when it is solved by a lemma learned or pushed by another task. If the algorithm terminates, then the conjunction of F_∞ is inductive according to the underlying SMT solver.

Our algorithm is parameterized by the logic used for inductive generalization, and thus the form of the invariant. We support universal, EPR, and full first-order logic (FOL) modes. Universal mode restricts the matrices to clauses, and

[6] Successor to CVC4 [1].

```
def P-FOL-IC3():
    $F_0$ = init ∪ safety;
    push();
    start LEARNING(), HEURISTIC();
    wait for invariant;
def MULTIBLOCK($\ell$: lemma, $s$: state, $i$):
    $S = \{s\}$;
    while not timed out:
        $p$ = IG($S$, $i$);
        speculatively add $p$ to frame $i$;
        $s', i'$ = to-block($\ell$);
        remove $p$ from frame $i$;
        if $i = i'$:
            add $s'$ to $S$;
        else:
            break
    add $p$ to frame $i$;
    push();

def LEARNING():
    while true:
        $s, i$ = to-block(safety);
        MULTIBLOCK(safety, $s$, $i$);
def HEURISTIC():
    while true:
        $\ell$ = random lemma before
          safety;
        $s, i$ = to-block($\ell$);
        if $i = 0$:
            mark $s$ reachable;
            mark bad lemmas;
        else:
            MULTIBLOCK($\ell$, $s$, $i$);
```

Fig. 2. Pseudocode for our proposed inference algorithm, P-FOL-IC3.

considers predecessors of superstructures when computing to-block() (as in [18]). EPR mode also takes as input the set of allowed edges. In FOL mode, there are no restrictions on the prefix.

6 Evaluation

We evaluate our algorithm and compare with prior approaches on a benchmark of invariant inference problems. We discuss the benchmark, our experimental setup, and the results.

6.1 Invariant Inference Benchmark

Our benchmark is composed of invariant inference problems from prior work on distributed protocols [29,28,7,27,30,2,9], written in or translated to the mypyvy tool's input language [26]. Our benchmark contains a total of 30 problems (Table 1), ranging from simple (toy-consensus, firewall) to complex (stoppable-paxos-epr, bosco-3t-safety). Some problems admit invariants that are purely universal, and others use universal and existential quantifiers, with some in EPR. All our examples are safe transition systems with a known human-written invariant.

6.2 Experimental Setup

We compare our algorithm to the techniques SWISS [14], IC3PO [11,12], fol-ic3 [19], and PDR$^\forall$ [18]. We performed our experiments on a 56-thread machine with 64 GiB of RAM, with each experiment restricted to 16 hardware threads, 20GiB of RAM, and a 6 hour time limit.[7] To account for noise caused by randomness in seed selection, we ran each algorithm 5 times and report the number

[7] Specifically, an dual-socket Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz.

Table 1. Experimental results, giving both the median wall-clock time (seconds) of run time and the number of trials successful, out of five. If there were less than 3 successful trials, we report the slowest successful trial, indicated by (>). A dash (-) indicates all trials failed or timed out after 6 hours (21600 seconds). A blank indicates no data.

Example	EPR	**Our**	#	Swiss	#	IC3PO	#	fol-ic3	#	PDR$^\forall$	#
lockserv	∀	19	5	9573	4	5	5	7	5	6	5
toy-consensus-forall	∀	4	5	22	5	4	5	11	5	4	5
ring-id	∀	7	5	192	5	81	5	28	5	20	5
sharded-kv	∀	8	5	17291	5	4	5	19	5	6	5
ticket	∀	23	5	-	0	-	0	240	5	22	5
learning-switch	∀	76	5	1744	4	29	5	-	0	94	5
consensus-wo-decide	∀	50	5	52	5	6	5	33	5	29	5
consensus-forall	∀	1908	5	80	5	15	5	1125	5	104	5
cache	∀	2492	4	-	0	3906	5	-	0	2628	5
paxos-forall	∀	885	5	-	0	-	0	-	0	555	5
flexible-paxos-forall	∀	1961	5	-	0	1654	5	-	0	423	5
stoppable-paxos-forall	∀	7779	5	-	0	-	0	-	0	-	0
fast-paxos-forall	∀	-	0	-	0	-	0	-	0	20176	3
vertical-paxos-forall	∀	-	0	-	0	-	0	-	0	-	0
firewall	–	4	5	-	0	3	5	9	5		
sharded-kv-no-lost-keys	✓	4	5	9	5	4	5	5	5		
toy-consensus-epr	✓	4	5	10	5	4	5	49	5		
ring-id-not-dead	–	19	5	-	0	-	0	221	3		
consensus-epr	✓	37	5	57	5	28	5	-	0		
client-server-ae	✓	4	5	11	5	4	5	442	5		
client-server-db-ae	–	16	5	46	5	37	5	6639	4		
hybrid-reliable-broadcast	–	178	5	-	0	-	0	937	5		
paxos-epr	✓	920	5	14332	4	-	0	-	0		
flexible-paxos-epr	✓	418	5	4928	5	-	0	-	0		
multi-paxos-epr	✓	4272	4	-	0	-	0	-	0		
stoppable-paxos-epr	✓	>18297	2	-	0	-	0	-	0		
fast-paxos-epr	✓	9630	3	-	0	-	0	-	0		
vertical-paxos-epr	✓	-	0	-	0	-	0	-	0		
block-cache-async	–	-	0	-	0	-	0	-	0		
bosco-3t-safety	✓	>11019[1]	1	-	0	-	0	-	0		

[1] With EPR push down enabled.

of successes and the median time. PDR$^\forall$, IC3PO, and fol-ic3 are not designed to use parallelism, while Swiss and our technique make use of parallelism. For IC3PO, we use the better result from the two implementations [11] and [12], and give reported results for those we could not replicate. For our technique, we ran the tool in universal-only, EPR, or full FOL mode as appropriate. For k-pDNF, we use $k = 1$ for universal prefixes and $k = 3$ otherwise.

6.3 Results and Discussion

We present the results of our experiments in Table 1. In general, for examples that converge with both prior approaches and our technique, we match or exceed existing results, with significant performance gains for some problems such as client-server-db-ae relative to the previous separation-based approach. Along with other techniques, we solve paxos-epr and flexible-paxos-epr, which are the simplest variants of Paxos in our benchmark, but nonetheless represents a significant jump in complexity over the examples solved by the prior generation of PDR/IC3 techniques. Paxos and its variants are notable for having invariants

Table 2. Ablation study. Columns are interpreted as in Table 1.

Example	**Our**	#	No pDNF	#	No EPR	#	No Inc. SMT	#	No Gen.	#
lockserv	19	5					34	5	13	5
toy-consensus-forall	4	5					5	5	4	5
ring-id	7	5					11	5	13	5
sharded-kv	8	5					11	5	7	5
ticket	23	5					42	5	21	5
learning-switch	76	5					338	5	288	5
consensus-wo-decide	50	5					50	5	51	5
consensus-forall	1908	5					2154	5	558	5
cache	2492	4					>16826	2	13116	5
paxos-forall	885	5					1071	5	10488	4
flexible-paxos-forall	1961	5					1014	5	>4168	2
stoppable-paxos-forall	7779	5					2820	5	>18727	1
fast-paxos-forall	-	0					>16573	1	-	0
vertical-paxos-forall	-	0					-	0	-	0
firewall	4	5	4	5			4	5	4	5
sharded-kv-no-lost-keys	4	5	4	5	4	5	5	5	4	5
toy-consensus-epr	4	5	5	5	5	5	5	5	5	5
ring-id-not-dead	19	5	37	5			44	5	52	5
consensus-epr	37	5	126	5	724	5	45	5	233	5
client-server-ae	4	5	3	5	4	5	4	5	4	5
client-server-db-ae	16	5	13	5			20	5	10	5
hybrid-reliable-broadcast	178	5	98	5			173	5	629	5
paxos-epr	920	5	10135	4	>2895	1	609	5	3201	5
flexible-paxos-epr	418	5	13742	3	-	0	775	5	799	5
multi-paxos-epr	4272	4	>15176	1	-	0	15854	3	7326	4
stoppable-paxos-epr	>18297	2	-	0	-	0	>20659	1	>11946	1
fast-paxos-epr	9630	3	-	0	-	0	8976	3	>20871	2
vertical-paxos-epr	-	0	-	0	-	0	-	0	-	0
block-cache-async	-	0	-	0			-	0	>20038	2
bosco-3t-safety	>11019	1	-	0	-	0	>8581	1	>16689	1

with two quantifier alternations (∀∃∀) and a maximum quantifier depth of 6 or 7. We uniquely solve multi-, fast-, and stoppable-paxos-epr, which add significant complexity in the number of sorts, symbols, and quantifier depth required. Due to variations in seeds and the non-determinism of parallelism, our technique was only successful in some trials, but these results nevertheless demonstrate that our technique is capable of solving these examples. Our algorithm is unable to solve vertical-paxos-epr, as this example requires a 7 quantifier formula that is very expensive for our IG solver.

For universal-only examples, our algorithm is able to solve all but one of the examples[8] solved by other techniques, and is able to solve one that others cannot. In some cases (e.g. consensus-forall), our solution is slower than other approaches, but on the whole our algorithm is competitive in a domain it is not specialized for. In addition, we significantly outperform the existing separation-based algorithm (fol-ic3) by solving several difficult examples (cache, paxos-forall).

6.4 Ablation Study

Table 2 presents an ablation study investigating effect of various features of our technique. The first column of Table 2 repeats the full algorithm results, and the remaining columns report the performance with various features disabled

[8] fast-paxos-forall, which is solved by our technique in the ablation study, albeit rarely.

Table 3. Parallel vs sequential comparison. Each of 5 trials ran with 3 or 48 hour timeouts, respectively. The number of successes, and the average number of IG queries in each trial (including failed ones) are given.

	Successes		IG Queries	
Example	Par.	Seq.	Par.	Seq.
paxos-epr	5	5	61	76
flexible-paxos-epr	5	5	64	72
multi-paxos-epr	3	1	67	84

individually. The most important individual contributions come from k-pDNF matrices and EPR. Using a 5-clause CNF instead of pDNF matrix (No pDNF) causes many difficult examples to fail and some (e.g., flexible-paxos-epr) to take significantly longer even when they do succeed.[9] Similarly, using full FOL mode instead of EPR (No EPR) leads to timeouts for all but the simplest Paxos variants. Incremental SMT queries (No Inc. SMT) make the more difficult Paxos variants, and the universal cache example, succeed much more reliably. Multi-block generalization (No Gen.) makes many problems faster or more reliable, but disabling it allows block-cache-async to succeed.

To isolate the benefits of parallelism, we ran several examples in both parallel and serial mode with a proportionally larger timeout (Table 3). In both modes we use a single prefix category containing all prefixes, with the same static order over prefixes.[10] Beyond the wall-clock speedup, the parallel IG algorithm affects the quality of the learned lemmas, that is, how well they generalize and avoid overfitting. To estimate the quality of generalization, we count the total number of IG queries performed by each trial and report the average over the five trials. In all examples, the parallel algorithm learns fewer lemmas overall, which suggests it generalizes better. We attribute this improved generalization to the implicit bias towards lemmas that are faster to discover. For the more complicated example (multi-paxos-epr), this difference has an impact on the success rate.

7 Related Work

Extensions of PDR/IC3. The PDR/IC3 [3,5] algorithm has been very influential as an invariant inference technique, first for hardware (finite state) systems and later for software (infinite state). There are multiple extensions of PDR/IC3 to infinite state systems using SMT theories [16,20]. [18] extended PDR/IC3 to universally quantified first-order formulas using the model-theoretic notion of *diagrams*. [13] applies PDR/IC3 to find universally quantified invariants over arrays and also to manage quantifier instantiation. Another extension of PDR/IC3 for universally quantified invariants is [23], where a quantified invariant is generalized from an invariant of a bounded, finite system. This technique of generalization from a bounded system has also been extended to quantifiers with al-

[9] With a single clause, there is no difference between CNF and k-pDNF so results are only given for existential problems.

[10] To make the comparison cleaner, we also disabled multi-block generalization.

ternations [11]. Recently, [31] suggested combining synthesis and PDR/IC3, but they focus on word-level hardware model checking and do not support quantifier alternations. Most of these works focus on quantifier-free or universally quantified invariants. In contrast, we address unique challenges that arise when supporting lemmas with quantifier alternations.

The original PDR/IC3 algorithm has also been extended with techniques that use different heuristic strategies to find more invariants by considering additional proof goals and collecting reachable states [15,17]. Our implementation benefits from some of these heuristics, but our contribution is largely orthogonal as our focus is on inductive generalization of quantified formulas. Generating lemmas from multiple states, similar to multi-block generalization, was explored in [21].

[24] suggests a way to parallelize PDR/IC3 by combining a portfolio approach with problem partitioning and lemma sharing. Our parallelism is more tightly coupled into PDR/IC3, as we parallelize the inductive generalization procedure.

Quantified Separation. Quantified separation [19] was recently introduced as a way to find quantified invariants with quantifier alternations. While [19] introduced a way to combine separation and PDR/IC3, it has limited scalability and cannot find the invariants of complex protocols such as Paxos. Our work here is motivated by these scalability issues. In contrast to [19], our technique is able to find complex invariants by avoiding expensive but useless areas of the search space using a breadth-first strategy and a multi-dimensional inductive bias. While [19] searches for quantified lemmas in CNF, we introduce and use k-term pDNF. k-term pDNF can express the necessary lemmas of many distributed protocols more succinctly, resulting in better scalability.

Synthesis-Based Approaches to Invariant Inference. Synthesis is a common approach for automating invariant inference. ICE [10] is a framework for learning inductive invariants from positive, negative, and implication constraints. Our use of separation is similar, but it is integrated into PDR/IC3's inductive generalization, so unlike ICE we find invariants incrementally.

Enumeration-Based Approaches. Another approach is to use enumerative search, for example [6], which only supports universal quantification. Enumerative search has been extended to quantifier alternations in [14], which is able to infer the invariants of complex protocols such as some Paxos variants.

8 Conclusion

We have presented an algorithm for quantified invariant inference that combines separation and inductive generalization. Our algorithm uses a breadth-first strategy to avoid regions of the search space that are expensive. We also explore a new syntactic form that is well-suited for lemmas with alternations. We show via a large scale experiment that our algorithm advances the state of the art in quantified invariant inference with alternations, and finds significantly more invariants on difficult problems than prior approaches.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (Jul 2011), https://dl.acm.org/doi/10.5555/2032305.2032319, Snowbird, Utah
2. Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. pp. 245–266 (2019). https://doi.org/10.1007/978-3-030-25543-5_15
3. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), https://link.springer.com/chapter/10.1007/978-3-642-18275-4_7
4. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), https://dl.acm.org/citation.cfm?id=1792734.1792766
5. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. pp. 125–134 (2011), https://dl.acm.org/citation.cfm?id=2157675
6. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 259–277. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_14
7. Feldman, Y.M., Padon, O., Immerman, N., Sagiv, M., Shoham, S.: Bounded quantifier instantiation for checking inductive invariants. In: Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205. pp. 76–95. Springer-Verlag, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_5
8. Feldman, Y.M.Y., Padon, O., Immerman, N., Sagiv, M., Shoham, S.: Bounded quantifier instantiation for checking inductive invariants. Log. Methods Comput. Sci. **15**(3) (2019). https://doi.org/10.23638/LMCS-15(3:18)2019
9. Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M.: Inferring inductive invariants from phase structures. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 405–425. Springer International Publishing, Cham (2019), https://link.springer.com/chapter/10.1007/978-3-030-25543-5_23
10. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A Robust Framework for Learning Invariants. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 69–87. Springer International Publishing, Cham (2014), https://link.springer.com/chapter/10.1007/978-3-319-08867-9_5
11. Goel, A., Sakallah, K.: On symmetry and quantification: A new approach to verify distributed protocols. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NASA Formal Methods. pp. 131–150. Springer International Publishing, Cham (2021), https://link.springer.com/chapter/10.1007/978-3-030-76384-8_9

12. Goel, A., Sakallah, K.A.: Towards an automatic proof of Lamport's Paxos. In: 2021 Formal Methods in Computer Aided Design (FMCAD). pp. 112–122 (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_20
13. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11138, pp. 248–266. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_15
14. Hance, T., Heule, M., Martins, R., Parno, B.: Finding invariants of distributed systems: It's a small (enough) world after all. In: Mickens, J., Teixeira, R. (eds.) 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021. pp. 115–131. USENIX Association (2021), https://www.usenix.org/conference/nsdi21/presentation/hance
15. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: 2013 Formal Methods in Computer-Aided Design. pp. 157–164 (2013). https://doi.org/10.1109/FMCAD.2013.6679405
16. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7317, pp. 157–171. Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_13
17. Ivrii, A., Gurfinkel, A.: Pushing to the top. In: 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 65–72 (2015). https://doi.org/10.1109/FMCAD.2015.7542254
18. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. J. ACM **64**(1), 7:1–7:33 (Mar 2017). https://doi.org/10.1145/3022187
19. Koenig, J.R., Padon, O., Immerman, N., Aiken, A.: First-order quantified separators. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 703–717. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386018
20. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
21. Krishnan, H.G.V., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. pp. 101–125 (2020). https://doi.org/10.1007/978-3-030-53291-8_7
22. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (may 1998). https://doi.org/10.1145/279227.279229
23. Ma, H., Goel, A., Jeannin, J., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: incremental inference of inductive invariants for verification of distributed protocols. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 370–384. ACM (2019). https://doi.org/10.1145/3341301.3359651
24. Marescotti, M., Gurfinkel, A., Hyvärinen, A.E.J., Sharygina, N.: Designing parallel PDR. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 156–163. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102254

25. McMillan, K.L.: Lazy annotation revisited. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 243–259 (2014). https://doi.org/10.1007/978-3-319-08867-9_16
26. mypyvy repository. https://github.com/wilcoxjay/mypyvy
27. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158114
28. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–31 (Oct 2017). https://doi.org/10.1145/3140568
29. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 614–630. PLDI '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908118
30. Taube, M., Losa, G., McMillan, K.L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J.R., Woos, D.: Modularity for decidability of deductive verification with applications to distributed systems. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 662–677. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192414
31. Zhang, H., Gupta, A., Malik, S.: Syntax-guided synthesis for lemma generation in hardware model checking. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 325–349. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_15

LinSyn: Synthesizing Tight Linear Bounds for Arbitrary Neural Network Activation Functions*

Brandon Paulsen✉ and Chao Wang

University of Southern California, Los Angeles CA 90089, USA
{bpaulsen,wang626}@usc.edu

Abstract. The most scalable approaches to certifying neural network robustness depend on computing sound linear lower and upper bounds for the network's activation functions. Current approaches are limited in that the linear bounds must be handcrafted by an expert, and can be sub-optimal, especially when the network's architecture composes operations using, for example, multiplication such as in LSTMs and the recently popular *Swish* activation. The dependence on an expert prevents the application of robustness certification to developments in the state-of-the-art of activation functions, and furthermore the lack of tightness guarantees may give a false sense of insecurity about a particular model. To the best of our knowledge, we are the first to consider the problem of *automatically* synthesizing *tight* linear bounds for arbitrary n-dimensional activation functions. We propose the first fully automated method that achieves tight linear bounds while only leveraging the mathematical definition of the activation function itself. Our method leverages an efficient heuristic technique to synthesize bounds that are tight and *usually sound*, and then verifies the soundness (and adjusts the bounds if necessary) using the highly optimized branch-and-bound SMT solver, dReal. Even though our method depends on an SMT solver, we show that the runtime is reasonable in practice, and, compared with state of the art, our method often achieves 2-5X tighter final output bounds and more than quadruple certified robustness.

1 Introduction

Prior work has shown that neural networks are vulnerable to various types of (adversarial) perturbations, such as small l-norm bounded perturbations [39], geometric transformations [13, 22], and word substitutions [2]. Such perturbations can often cause a misclassification for any given input, which may have serious consequences, especially in safety critical systems. Certifying robustness to these perturbations has become an important problem as it can show the network does not exhibit these misclassifications, and furthermore previous work has shown that a given input feature's certified robustness can be a useful indicator to determine the feature's importance in the network's decision [34, 25].

* This work was partially funded by the U.S. National Science Foundation grants CNS-1813117 and CNS-1722710, and the U.S. Office of Naval Research (ONR) grant N00014-17-1-2896.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 357–376, 2022.
https://doi.org/10.1007/978-3-030-99524-9_19

Indeed, many approaches have been proposed for certifying the robustness of inputs to these perturbations. Previous work typically leverages two types of techniques: (1) fast and scalable, but approximate techniques [36, 15, 45, 34, 25], and (2) expensive but exact techniques that leverage some type of constraint solver [23, 24, 40]. Several works have also combined the two [37, 35, 43, 42]. The most successful approaches, in terms of scalability in practice, are built on top of the approximate techniques, which all depend on computing *linear bounds* for the non-linear activation functions.

However, a key limitation is that the linear bounds must be handcrafted and proven sound by experts. Not only is this process difficult, but also ensuring the tightness of the crafted bounds presents an additional challenge. Unfortunately, prior work has only crafted bounds for the most common activation functions and architectures, namely ReLU [43], sigmoid, tanh [36, 48, 46], the exp function [34], and some 2-dimensional activations found in LSTM networks [25]. As a result, existing tools for neural network verification cannot handle a large number of activation functions that are frequently used in practice. Examples include the *GELU* function [18], which is currently the activation function used in OpenAI's GPT [31], and the *Swish* function which has been shown to outperform the standard ReLU function in some applications [32] and, in particular, can reduce over-fitting in adversarial training [38]. In addition, these recently introduced activation functions are often significantly more complex than previous activation functions, e.g., we have $gelu(x) = 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$.

In this work, we study the problem of *efficiently* and *automatically* synthesizing *sound* and *tight* linear bounds for any *arbitrary activation function*. By *arbitrary activation function*, we mean *any* (non-linear) computable function $z = \sigma(x_1, \ldots, x_d)$ used inside a neural network with d input variables. By *sound* we mean, given an interval bound on each variable $x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d]$, the problem is to *efficiently* compute lower bound coefficients $c_1^l, c_2^l, \ldots, c_{d+1}^l$, and upper bound coefficients $c_1^u, c_2^u, \ldots, c_{d+1}^u$ such that the following holds:

$$\begin{gathered}\forall x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ c_1^l x_1 + c_2^l x_2 + \cdots + c_{d+1}^l \leq \sigma(x_1, \ldots, x_d) \leq c_1^u x_1 + c_2^u x_2 + \cdots + c_{d+1}^u\end{gathered} \tag{1}$$

By *automatically*, we mean that the above is done using only the definition of the activation function itself. Finally, by *tight*, we mean that some formal measure, such as the volume above/below the linear bound, is minimized/maximized.

We have developed a new method, named LinSyn, that can *automatically* synthesize tight linear bounds for *any arbitrary* non-linear activation function $\sigma(\cdot)$. We illustrate the flow of our method on the left-hand side of Fig. 1. As shown, LinSyn takes two inputs: a definition of the activation function, and an interval for each of its inputs. LinSyn outputs linear coefficients such that Equation 1 holds. Internally, LinSyn uses sampling and an LP (linear programming) solver to synthesize candidate lower and upper bound coefficients. Next, it uses an efficient local minimizer to compute a good estimate of the offset needed to ensure soundness of the linear bounds. Since the candidate bounding functions constructed in this manner may still be unsound, finally, we use a highly optimized branch-and-bound nonlinear SMT solver, named dReal [14], to verify

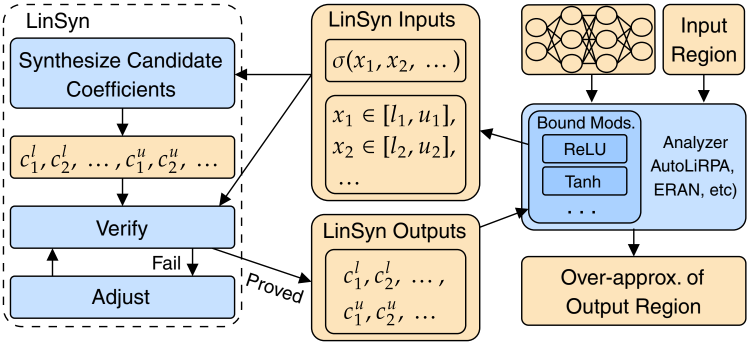

LinSyn
Synthesize Candidate Coefficients
$c_1^l, c_2^l, \ldots, c_1^u, c_2^u, \ldots$
Verify
Fail
Adjust
Proved
LinSyn Inputs
$\sigma(x_1, x_2, \ldots)$
$x_1 \in [l_1, u_1]$, $x_2 \in [l_2, u_2]$, …
LinSyn Outputs
$c_1^l, c_2^l, \ldots,$ $c_1^u, c_2^u, \ldots$
Input Region
Bound Mods.
ReLU
Tanh
. . .
Analyzer AutoLiRPA, ERAN, etc)
Over-approx. of Output Region

Fig. 1. The overall flow of LinSyn.

the soundness of the linear bounds. Even though our new method involves the use of solvers and optimizers, the entire process typically takes less than 1/100th of a second per pair of bounds.

Fig. 1 also illustrates how LinSyn fits in with existing neural network verification frameworks, such as ERAN [1], and AutoLiRPA [47]. These tools take as input a neural network, and a region of the neural networks input space, and compute an over-approximation of the neural network's outputs. Internally, these frameworks have modules that compute linear bounds for a specific activation functions. LinSyn is a one-size-fits-all drop-in replacement for these modules that are invoked at runtime whenever a linear bound of a non-linear activation function is needed.

Our method differs from these existing frameworks because a user (usually an expert in neural network verification) must provide hand-crafted, sound linear bounds for the activation functions of a neural network. However, to date, they only support the previously mentioned activation functions. We note however that the recent framework AutoLiRPA supports binary operations (namely addition, subtraction, multiplication, and division) as "activation functions". Thus, while it's not explicitly designed to handle complex activations, it has the ability to by decomposing, e.g., $gelu(x)$ into operations that it supports, and then combining them. In contrast, LinSyn bounds the activation function *as a whole*, which we will show produces much tighter linear bounds.

We have implemented our method in tool called LinSyn, and evaluated it on benchmarks in computer vision and natural language processing (NLP). Our evaluation shows that we can obtain final output bounds often 2-5X tighter than the most general tool [47], thus allowing us to drastically increase certified robustness. In addition, our tool achieves accuracy equal to or better than the handcrafted LSTM bounds of Popqorn [25], which is currently the most accurate tool for analyzing LSTM-based NLP models, at a comparable runtime.

To summarize, this paper makes the following contributions:

- We propose the first method for automatically synthesizing tight linear bounds for arbitrary activation functions.

- We implement our approach in a tool called LinSyn, and integrate it as a bounding module into the AutoLiRPA framework, thus producing a neural network verification tool that can theoretically compute tight linear bounds for any arbitrary activation function.
- We extensively evaluate our approach and show it outperforms state-of-the-art tools in terms of accuracy and certified robustness by a large margin.

The rest of this paper is organized as follows. First, we provide the technical background in Section 2. Then, we present our method for synthesizing the linear bounds in Section 3 and our method for verifying the linear bounds in Section 4. Next, we present the experimental results in Section 5. We review the related work in Section 6 and, finally, give our conclusions in Section 7.

2 Preliminaries

In this section, we define the neural network verification problem, and illustrate both how state-of-the-art verification techniques work, and their limitations.

2.1 Neural Networks

Following conventional notation, we refer to matrices with capital bold letters (e.g. $\mathbf{W} \in \mathbb{R}^{n \times m}$), vectors as lower case bold letters (e.g. $\mathbf{x} \in \mathbb{R}^n$), and scalars or variables with lower case letters (e.g. $x \in \mathbb{R}$). Slightly deviating from the convention, we refer to a set of elements with capital letters (e.g. $X \subseteq \mathbb{R}^n$).

We consider two types of networks in our work: feed-forward and recurrent. We consider a feed-forward neural network to be a (highly) non-linear function $f : \mathbb{X} \to \mathbb{Y}$, where $\mathbb{X} \subseteq \mathbb{R}^n$ and $\mathbb{Y} \subseteq \mathbb{R}^m$. We focus on neural network *classifiers*. For an input $\mathbf{x} \in \mathbb{X}$, each element in the output $f(\mathbf{x})$ represents a score for a particular class, and the class associated with the largest element is the chosen class. For example, in image classification, $\mathbb{X}$ would be the set of all images, each element of an input $\mathbf{x} \in \mathbb{X}$ represents a pixel's value, and each element in $\mathbb{Y}$ is associated with a particular object that the image might contain.

In feed-forward neural networks the output $f(\mathbf{x})$ is computed by performing a series of affine transformations, i.e., multiplying by a weight matrix, followed by application of an activation function $\sigma(\cdot)$. Formally, a neural network with l layers has l two-dimensional weight matrices and l one-dimensional bias vectors $\mathbf{W_i}, \mathbf{b_i}$, where $i \in 1..l$, and thus we have $f(\mathbf{x}) = \mathbf{W_l} \cdot \sigma(\mathbf{W_{l-1}} \cdots \sigma(\mathbf{W_1} \cdot \mathbf{x} + \mathbf{b_1}) \cdots + \mathbf{b_{l-1}}) + \mathbf{b_l}$, where $\sigma(\cdot)$ is the activation function applied element-wise to the input vector. The default choice of activation is typically the sigmoid $\sigma(x) = 1/(1 + e^{-x})$, tanh , or ReLU function $\sigma(x) = max(0, x)$, however recent work [18, 32, 31] has shown that functions such as $gelu(x)$ and $swish(x) = x \times sigmoid(x)$ can have better performance and desirable theoretical properties.

Unlike feed-forward neural networks, recurrent neural networks receive a sequence of inputs $[\mathbf{x^{(1)}}, \ldots, \mathbf{x^{(t)}}]$, and the final output of f on $\mathbf{x_t}$ is used to perform the classification of the whole sequence. Recurrent neural networks are *state-ful*, meaning they maintain a state vector that contains information about inputs previously given to f, which also gets updated on each call to f. In particular, we focus on *long short-term memory* (LSTM) networks, which have seen wide adoption in natural language processing (NLP) tasks due to their sequential nature. For LSTMs trained for NLP tasks, the network receives a sequence of *word embeddings*. A word embedding is an n-dimensional vector that is associated with a particular word in a (natural) language. The distance between word embeddings carries semantic significance – two word embeddings that are close to each other in $\mathbb{R}^n$ typically have similar meanings or carry a semantic relatedness (e.g. *dog* and *cat* or *king* and *queen*), whereas unrelated words typically are farther apart.

LSTM networks further differ from feed-forward networks in that their internal activation functions are *two*-dimensional. Specifically, we have the following two activation patterns: $\sigma_1(x) \times \sigma_2(y)$ and $x \times \sigma_1(y)$. The default choices are $\sigma_1(x) = sigmoid(x)$, and $\sigma_2(x) = tanh(x)$. However, we can swap σ_1 with any function with output range bounded by $[0, 1]$, and swap σ_2 with any function with output range bounded by $[-1, 1]$. Indeed, prior work [16] has shown that $\sigma_1(x) = 1 - e^{e^{-x}}$ can achieve better results in some applications.

2.2 Neural Network Verification

A large number of problems in neural network verification can be phrased as the following: given an input region $X \subseteq \mathbb{X}$, compute an over-approximation Y, such that $\{f(\mathbf{x}) \mid \mathbf{x} \in X\} \subseteq Y \subseteq \mathbb{Y}$. Typically X and Y are hyper-boxes represented by an interval for each of their elements. A common problem is to prove that a point $\mathbf{x} \in \mathbb{X}$ is *robust*, meaning that small perturbations will not cause an incorrect classification. In this case, X is the set of all perturbed versions of $\mathbf{x}$, and to prove robustness, we check that the element of the correct class in Y has a lower bound that is greater than the upper bound of all other elements.

We illustrate a simple verification problem on the neural network shown in Fig. 2. The network has two inputs, x_1, x_2, and two outputs x_7, x_8 which represent scores for two different classes. We refer to the remaining hidden neurons as $x_i, i \in 3..6$. Following prior work [36], we break the affine transformation and application of the activation function into two separate neurons, and the neurons are assumed to be ordered such that, if x_i is in a layer before x_j, then $i < j$. For simplicity, in this motivating example, we let $\sigma(x) = max(0, x)$ (the ReLU function). We are interested in proving that the region $x_1 \in [-1, 1], x_2 \in [-1, 1]$ always maps to the first class, or in other words, we want to show that the lower bound of x_7 is greater than the upper bound x_8.

2.3 Existing Methods

The most scalable approaches (to date) for neural network verification are based on linear bounding and back-substitution [47], also referred to as abstract interpretation in the polyhedral abstract domain [36] or symbolic interval analysis [43] in prior work.

For each neuron x_j in the network, these approaches compute a concrete lower and upper bound l_j, u_j, and a linear lower and upper bound in terms of the previous layer's neurons. The linear bounds (regardless of the choice of $\sigma(\cdot)$)

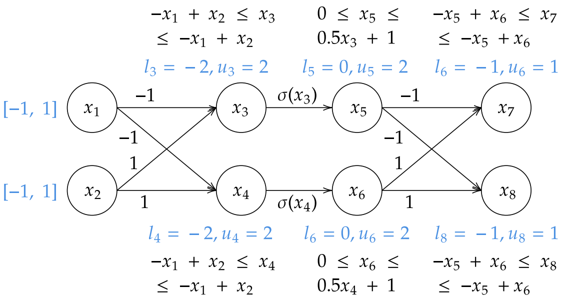

$-x_1 + x_2 \leq x_3 \leq -x_1 + x_2$
$0 \leq x_5 \leq 0.5x_3 + 1$
$-x_5 + x_6 \leq x_7 \leq -x_5 + x_6$
$l_3 = -2, u_3 = 2$
$l_5 = 0, u_5 = 2$
$l_6 = -1, u_6 = 1$
[−1, 1]
[−1, 1]
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
−1
−1
1
1
$\sigma(x_3)$
$\sigma(x_4)$
−1
−1
1
1
$l_4 = -2, u_4 = 2$
$l_6 = 0, u_6 = 2$
$l_8 = -1, u_8 = 1$
$-x_1 + x_2 \leq x_4 \leq -x_1 + x_2$
$0 \leq x_6 \leq 0.5x_4 + 1$
$-x_5 + x_6 \leq x_8 \leq -x_5 + x_6$

Fig. 2. Example of neural network verification.

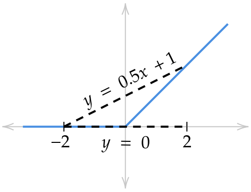


Fig. 3. Linear bounds for ReLU activation.

have the following form: $\sum_{i=0}^{j-1} x_i \cdot c_i^l + c_j^l \leq x_j \leq \sum_{i=0}^{j-1} x_i \cdot c_i^u + c_j^u$. The bounds are computed in a forward, layer-by-layer fashion which guarantees that any referenced neurons will already have a bound computed when back-substitution is performed.

To obtain the concrete bounds l_j, u_j for a neuron x_j, the bounds of any non-input neurons are recursively substituted into the linear bounds of x_j until only input nodes $x_1, ..., x_n$ remain. Finally, the concrete input intervals are substituted into the bound to obtain l_j, u_j.

Example We illustrate on the two-layer network in Fig. 2 for the previously defined property. We trivially have $l_1 = l_2 = -1$, $u_1 = u_2 = 1$, $-1 \leq x_1 \leq 1$, and $-1 \leq x_2 \leq 1$. We then compute linear bounds for x_3, x_4 in terms of previous layer's neurons x_1, x_2. We multiply x_1, x_2 by the edge weights, obtaining $-x_1+x_2$ as the lower and upper bound for both of x_3 and x_4. Since this bound is already in terms of the input variables, we substitute the concrete bounds into this equation and obtain $l_3 = l_4 = -2$ and $u_3 = u_4 = 2$.

Next, we need to compute the linear bounds for $x_5 = \sigma(x_3)$ and $x_6 = \sigma(x_4)$ after applying the activation function. Solving this challenge has been the focus of many prior works. There are two requirements. First, they need to be *sound*. For example, for x_5 we need to find coefficients $c_1^l, c_2^l, c_1^u, c_2^u$ such that $c_1^l x_3 + c_2^l \leq \sigma(x_3) \leq c_1^u x_3 + c_2^u$ for all $x_3 \in [l_3, u_3]$, and similarly for x_6. Second, we want them to be *tight*. Generally, this means that volume below the upper bound is minimized, and volume below the lower bound is maximized.

As an example, prior work [36, 48] proposed the following sound and tight bound for $\sigma(x) = max(0, x)$:

$$\forall x_i \in [l_i, u_i] \ . \ \frac{u_i}{u_i - l_i} x_i + \frac{-l_i u_i}{u_i - l_i} \leq \sigma(x_i) \leq \begin{cases} 0 & -l_i \geq u_i \\ x_i & -l_i < u_i \end{cases}$$

We illustrate the bound for x_5 in Fig. 3. After computing this bound, we recursively substitute variables in the bounds of x_5 with the appropriate bound, and compute l_5, u_5. The process then repeats for x_6, followed by x_7 and x_8. We then check $l_7 > u_8$ to verify the property, which fails in this case.

2.4 Limitations of Existing Methods

Current approaches only support a limited number of activation functions, and designing linear bounds for new activation functions often requires a significant amount of effort even for a domain expert. For example, handcrafted sound and tight linear bounds for activation functions such as ReLU, sigmoid, and tanh [36, 45, 48, 46, 44, 43], convolution layers and pooling operations [6], the two-dimensional activations found in LSTMs [25, 33], and those in transformer networks [34] are worthy of publication. Furthermore, even bounds that are hand-crafted by experts are not always tight. For example, a recent work [46] was able to nearly triple the precision of previous state-of-the-art sigmoid and tanh linear bounds simply by improving tightness.

To the best of our knowledge, AutoLiRPA [47] is the only tool that has the ability to handle more complex activation functions, though it was not originally designed for this. It can do so by decomposing them into simpler operations, and then composing the bounds together. We illustrate with $swish(x) = x \times sigmoid(x)$, where $x \in [-1.5, 5.5]$. AutoLiRPA would first bound $sigmoid(x)$ over the region $[-1.5, 5.5]$, resulting in the bound $.11x + .35 \leq sigmoid(x) \leq .22x + .51$. For the left-hand side of the function, we trivially have $x \leq x \leq x$. AutoLiRPA would then bound a multiplication $y \times z$, where in this case $y = x$ and $z = sigmoid(x)$, resulting in the final bound $-.15x - .495 \leq x \times sigmoid(x) \leq 0.825x + .96$. We illustrate this bound in Fig. 4, and we provide bounds computed by LinSyn as a comparison point. LinSyn provides a slightly better upper bound, and a significantly better lower bound. The reason for the looseness is because when AutoLiRPA bounds $sigmoid(x)$, it necessarily accumulates some approximation error because it is approximating the behavior of a non-linear function with linear bounds. The approximation error effectively "loses some information" about about its input variable x. Then, when bounding the multiplication operation, it has partially lost the information that y and z are related (i.e. they are both derived from x). In contrast, LinSyn overcomes this issue by considering $swish(x)$ as a whole. We explain how in the following sections.

3 Synthesizing the Candidate Linear Bounds

In this section, we describe our method for synthesizing candidate, possibly unsound linear bounds.

3.1 Problem Statement and Challenges

We assume we are given a d-dimensional activation function $z = \sigma(x_1, ..., x_d)$, and an input interval $x_i \in [l_i, u_i]$ for each $i \in \{1..d\}$. Our goal is to synthesize linear coefficients c_i^l, c_i^u, where $i \in \{1..d+1\}$ that are sound, meaning that the following condition holds:

$$\begin{gathered}\forall x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots, x_d \in [l_d, u_d] \\ c_1^l x_1 + c_2^l x_2 + \cdots + c_{d+1}^l \leq \sigma(x_1, x_2, \dots) \leq c_1^u x_1 + c_2^u x_2 + \cdots + c_{d+1}^u\end{gathered} \tag{2}$$

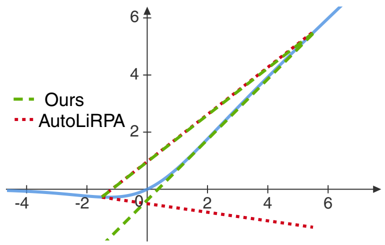


Fig. 4. Bounds computed by LinSyn and AutoLiRPA for $swish(x)$, $x \in [-1.5, 5.5]$.

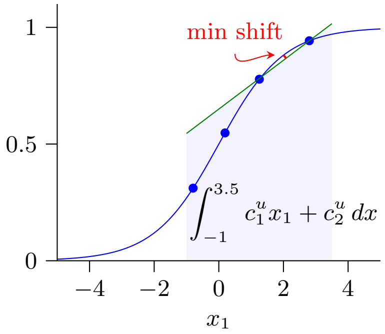


Fig. 5. Candidate plane synthesis.

In addition, we want to ensure that the bounds are *tight*. The ideal definition of tightness would choose linear bounds that maximize the precision of the overall analysis, for example minimizing the width of the output neuron's intervals. Unfortunately, such a measure would involve all of the neurons of the network, and so is impractical to compute. Instead, the common practice is to settle for tightness that's local to the specific neuron we are bounding.

Informally, we say a bound is *tight* if the volume below the upper bound is minimized, and volume below the lower bound is maximized. Prior work [48, 36, 25] has found this to be a good heuristic[1]. Formally, volume is defined as the following integral: $\int_{l_1}^{u_1} \cdots \int_{l_d}^{u_d} \sum_{i=1}^{d} c_i^u x_i + c_{d+1}^u \; dx_1 \ldots dx_d$ which, for the upper bound, should be minimized subject to Equation 2. This integral has the following closed-form solution:

$$\sum_{i=0}^{d}\left[\frac{1}{2}c_i \times \prod_{j=0}^{d}\left(u_i^{1+\mathbf{1}_{i=j}} - l_i^{1+\mathbf{1}_{i=j}}\right)\right] + c_{d+1} * \prod_{i=0}^{d}(u_i - l_i) \tag{3}$$

where $\mathbf{1}_{i=j}$ is the (pseudo Boolean) indicator function that returns 1 when its predicate is true. We omit the proof, but note that the above expression can be derived inductively on d. Also note that, since each l_i, u_i are concrete, the above expression is linear in terms of the coefficients, which will be advantageous in our approach below.

While recent approaches in solving non-linear optimization problems [26, 8] could directly minimize Equation 3 subject to Equation 2 in one step, we find the runtime to be very slow. Instead, we adopt a two-step approach that first uses efficient procedures for computing candidate coefficients that are almost sound (explained in this section), and second, only calls an SMT solver when necessary to verify Equation 2 (explained in the next section). We illustrate the approach on a concrete example.

[1] We also experimented with minimizing the volume between the linear bound and the activation function, which gave almost identical results.

3.2 Synthesizing Candidate Bounds

The first step in our approach computes candidate coefficients for the linear bound. In this step we focus on satisfying the tightness requirement, while making a best effort for soundness. We draw inspiration from prior work [33, 3] that leverages sampling to estimate the curvature of a particular function, and then uses a linear programming (LP) solver to compute a plane that is sound. However, unlike prior work which targeted a fixed function, we target arbitrary (activation) functions, and thus these are special cases of our approach.

The constraints of the LP are determined by a set of sample points $S \subset \mathbb{R}^d$. For the upper bound, we minimize Equation 3, subject to the constraint that the linear bound is above $\sigma(\cdot)$ at the points in S. Using $\mathbf{s}_i$ to refer to the i^{th} element of the vector $\mathbf{s} \in S$, the linear program we solve is:

$$\text{minimize Equation (3)} \;\; \text{subject to} \bigwedge_{\mathbf{s}\in S} c_1\mathbf{s}_1 + c_2\mathbf{s}_2 + \cdots + c_{d+1} \geq \sigma(\mathbf{s}) \tag{4}$$

We generate S by sampling uniformly-spaced points over the input intervals.

Example We demonstrate our approach on the running example illustrated in Fig. 5. For the example, let $\sigma(x_1) = \frac{1}{1+e^{-x_1}}$ (the sigmoid function, shown as the blue curve), where $x_1 \in [-1, 3.5]$. We focus only on the upper bound, but the lower bound is computed analogously.

Plugging in the variables into Equation 3, the objective of the LP that we minimize is: $\displaystyle\int_{-1}^{3.5} c_1^u x_1 + c_2^u \, dx_1 = 6.625c_1^u + 4.5c_2^u$ which is shown as the shaded region in Fig. 5.

We sample the points $S = \{-1, 0.25, 1.5, 2.75\}$, resulting in the following four constraints: $-c_1 + c_2 \geq \sigma(-1) \wedge 0.25c_1 + c_2 \geq \sigma(0.25) \wedge 1.5c_1 + c_s \geq \sigma(1.5) \wedge 2.75c_1 + c_2 \geq \sigma(2.75)$. Solving the LP program results in $c_1 = 0.104, c_2 = 0.649$, which is illustrated by the green line in Fig. 5.

4 Making the Bound Sound

In this section, we present our method for obtaining soundness because the candidate bounds synthesized in the previous section may not be sound. Here, we focus only on making the upper bound sound, but note the procedure for the lower bound is similar.

4.1 Problem Statement and Challenges

We are given the activation function $\sigma(\cdot)$, the input intervals $x_i \in [l_i, u_i]$, and the candidate coefficients $c_1, c_2, \ldots, c_{d+1}$. The goal is to compute an upward shift, if needed, to make the upper bound sound. First, we define the violation of the upper bound as:

$$v(x_1, x_2, \ldots, x_d) := c_1^u x_1 + c_2^u x_2 + \cdots + c_{d+1}^u - \sigma(x_1, x_2, \ldots, x_d) \tag{5}$$

A negative value indicates the upper bound is not sound. We then need to compute a lower bound on $v(\cdot)$, which we term v_l. Then the equation we pass to the verifier is:

$$\begin{gathered}\forall x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ v(x_1, x_2, \ldots, x_d) + (-v_l) \geq 0\end{gathered} \tag{6}$$

Expanding $v(\cdot)$ with its definition in the above equation results in the soundness definition of Equation 2. Thus, if the verifier proves Equation 6, then shifting the upper bound upward by $-v_l$ ensures its soundness. For our running example, the quantity v_l is shown by the red line in Fig. 5.

This problem is non-trivial because finding a solution for v_l requires a search for a sound global minimum/maximum of a function involving $\sigma(\cdot)$, which may be highly non-linear. State-of-the-art SMT solvers such as Z3 do not support all non-linear operations, and furthermore, since we assume arbitrary $\sigma(\cdot)$, the problem may even be (computationally) undecidable.

4.2 Verifying the Bound

We first assume we have a candidate (possibly unsound) v_l, and explain our verification method. To ensure decidability and tractability, we leverage the δ-*decision procedure* implemented by dReal [14]. To the best of our knowledge this is is the only framework that is decidable for all computable functions.

In this context, instead of verifying Equation 6, the formula is first negated thus changing it into an existentially quantified one, and then applying a δ-*relaxation*. Formally, the formula dReal attempts to solve is:

$$\begin{gathered}\exists x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ v(x_1, x_2, \ldots) + (-v_l) \leq \delta\end{gathered} \tag{7}$$

where δ is a small constant (e.g. 10^{-5}), which we explain in a moment. The above is formulated such that Equation 6 holds if (but not only if) there does *not* exist a solution to Equation 7.

Internally, dReal performs interval constraint propagation (ICP) on the left-hand side of Equation 7 over the intervals defined by each $[l_i, u_i]$ to compute an upper bound, and compares this upper bound with δ. If the upper bound is less than δ, then no solution exists (i.e., Equation 7 is unsatisfiable, and we have proven the original Equation 6 holds). Otherwise a solution *may* exist. In this case, dReal iteratively partitions the input space defined by the $[l_i, u_i]$ and repeats this process on each partition separately.

dReal stops partitioning either when it proves all partitions do not have solutions , or when a partition whose intervals all have width less than some ϵ is found. Here, ϵ is proportional to δ (i.e., smaller δ means smaller ϵ). In the latter case, dReal returns this partition as a "solution".

While Equation 6 holds if there does not exist a solution to Equation 7, the converse does not hold true both because of the error inherent in ICP, and because we "relaxed" the right-hand side of Equation 7. This means that δ controls the *precision* of the analysis. δ controls both the size of the false solution space, and determines how many times we will sub-divide the input space before giving up on proving Equation 7 to be unsatisfiable.

Practically, this has two implications for our approach. The first one is that our approach naturally inherits a degree of looseness in the linear bounds defined by δ. Specifically, we must shift our plane upward by δ in addition to the true v_l, so that dReal can verify the bound. The second is that we have to make a trade-off between computation and precision. While smaller δ will allow us to verify a tighter bound, it generally will also mean a longer verification time. In our experiments, we find that $\delta = 10^{-7}$ gives tight bounds at an acceptable runtime, though we may be able to achieve a shorter runtime with a larger δ.

4.3 Computing v_l

Now that we have defined how we can verify a candidate bound, we explain our approach for computing v_l. The implementation is outlined in Algorithm 1. Since failed calls to the verifier can be expensive, at lines 1-2, we first use a relatively cheap (and unsound) local optimization procedure to estimate the true v_l. While local optimization may get stuck in local minima, neural network activation functions typically do not have many local minima, so neither will $v(\cdot)$. We use L-BFGS-B [7], the bounded version of L-BFGS, to perform the optimization. At a high-level, L-BFGS-B takes as input $v(\cdot)$, the input bounds $x_i \in [l_i, u_i]$, and an initial guess $\mathbf{g} \in \mathbb{R}^d$ at the location of the local minimum. It then uses the Jacobian matrix (i.e., derivatives) of $v(\cdot)$ to iteratively move towards the local minimum (the Jacobian can be estimated using the finite differences method or provided explicitly – we use Mathematica [21] to obtain it). We find that sampling points uniformly in $v(\cdot)$ can usually find a good $\mathbf{g}$, and thus L-BFGS-B often converges in a small number of iterations. L-BFGS-B typically produces an estimate within 10^{-8} of the true value. To account for estimation error we add an additional 10^{-6}, plus $2 \times \delta$ to account for the δ-relaxation (line 3). Finally, we iteratively decrease v_l by a small amount (10^{-6}) until dReal verifies it (lines 4-9).

Going back to our motivating example, we would estimate v_l with a local minimizer, and then use dReal to verify the following:

$$\forall x_1 \in [-1, 3.5] \; . \; \sigma(x_1) \leq c_1^u x_1 + c_2^u + (-v_l) + 2 \times \delta + 10^{-6}$$

If verification fails, we iteratively decrease the value of v_l by 10^{-6}, and call dReal until the bound is verified. The final value of $c_1^u x_1 + c_2^u + (-v_l) + 2 \times \delta + 10^{-6}$ is the final sound upper bound.

4.4 On the Correctness and Generality of LinSyn

The full LinSyn procedure is shown in Algorithm 2. The correctness (i.e. soundness) of the synthesized bounds is guaranteed if the v_l returned by Algorithm 1 is a true lower bound on $v(\cdot)$. Since Algorithm 1 does not return until dReal verifies v_l at line 6, the correctness is guaranteed.

Both our procedure in Section 3 and L-BFGS-B require only black-box access to $\sigma(\cdot)$, so the only potential limit to the arbitrariness of our approach lies in

Algorithm 1: BoundViolation

```
Input: Activation $\sigma(x_1, x_2, \dots)$, Candidate Coefficients $c_1^u, c_2^u, \dots, c_{d+1}^u$,
       Input Bounds $x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots$, Jacobian $\nabla v$ (optional)
Output: Lower Bound on Violation $v_l$
$\mathbf{g} \leftarrow$ sample points on $v(x_1, x_2, \dots)$ and take minimum;
$v_l \leftarrow$ L-BFGS-B$(v(x_1, x_2, \dots), x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots, \mathbf{g}, \nabla v)$ ;
$v_l \leftarrow v_l - 10^{-6} - 2\delta$;
while True do
    // Call dReal
    if Equation 2 holds then
        return $v_l$;
    end
    $v_l \leftarrow v_l - 10^{-6}$;
end
```

Algorithm 2: SynthesizeUpperBoundCoefficients

```
Input: Activation $\sigma(x_1, x_2, \dots)$, Input Bounds $x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots$,
       Jacobian $\nabla v$ (optional)
Output: Sound Coefficients $c_1^u, c_2^u, \dots, c_{d+1}^u$
$c_1^u, c_2^u, \dots, c_{d+1}^u \leftarrow$ Sampling and LP procedure on $\sigma(x)$ over Input Bounds;
$v_l \leftarrow$ BoundViolation$(c_1^u, c_2^u, \dots, c_{d+1}^u, x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots, \nabla v)$;
$c_{d+1}^u \leftarrow c_{d+1}^u + (-v_l)$;
return $c_1^u, c_2^u, \dots, c_{d+1}^u$;
```

what elementary operations are supported by dReal. During our investigation, we did not find activations that use operations unsupported by dReal, however if an unsupported operation is encountered, one would only need to define an *interval extension* [28] for the operation, which can be done for any computable function.

5 Evaluation

We have implemented our method in a module called LinSyn, and integrated it into the AutoLiRPA neural network verification framework [47]. A user instantiates LinSyn with a definition of an activation function, which results in an executable software module capable of computing the sound linear lower and upper bounds for the activation function over a given input region. LinSyn uses Gurobi [17] to solve the LP problem described in Section 3, and dReal [14] as the verifier described in 4. In total, LinSyn is implemented in about 1200 lines of Python code.

5.1 Benchmarks

Neural Networks Our benchmarks are nine deep neural networks trained on the three different datasets shown below. In the following, a neuron is a node in the neural network where a linear bound must be computed, and thus the neuron counts indicate the number of calls to LinSyn that must be made.

- **MNIST:** MNIST is a dataset of hand-written integers labeled with the corresponding integer in the image. The images have 28x28 pixels, with each pixel taking a gray-scale value between 0 to 255. We trained three variants of a 4-layer CNN (convolutional neural network). Each takes as input a 28x28 = 784-dimensional input vector and outputs 10 scores, one for each class. In total, each network has 2,608 neurons – 1568, 784, and 256 in the first, second, and third layers, respectively.
- **CIFAR:** CIFAR is a dataset of RGB images from 10 different classes. The images have 32x32 pixels, with each pixel having an R, G, and B value in the range 0 to 255. We trained three variants of a 5-layer CNN. Each takes a 32x32x3 = 3072-dimensional input vector and outputs 10 scores, one for each class. In total, each network has 5376 neurons, 2048, 2048, 1024, and 256 neurons in the first, second, third, and fourth layers, respectively.
- **SST-2:** The Stanford Sentiment Treebank (SST) dataset consists of sentences taken from movie reviews that are human annotated with either positive or negative, indicating the sentiment expressed in the sentence. We trained three different variants of the standard LSTM architecture. These networks take as input a sequence 64-dimensional word embeddings and output 2 scores, one for positive and one for negative. Each network has a hidden size of 64, which works out to 384 neurons per input in the input sequence.

Activation Functions We experimented with the four activation functions as shown in Fig. 6. *GELU* and *Swish* were recently proposed alternatives to the standard ReLU function due to their desirable theoretical properties [18] such as reduced overfitting [38], and they have seen use in OpenAI's GPT [31] and very deep feed forward networks [32]. Similarly, *Hard-Tanh* is an optimized version of the common tanh function, while the *Log-Log* function [16] is a sigmoid-like function used in forecasting.

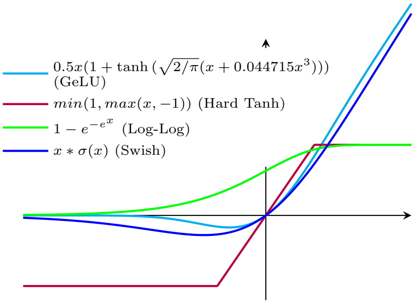


Fig. 6. Nonlinear activation functions.

The Verification Problem The verification problem we consider is to certify that an input is robust to bounded perturbations of magnitude ϵ, where ϵ is a small number. *Certifying* means proving that the classification result of the neural network does not change in the presence of perturbations. We focus on l_∞ robustness, where we take an input $\mathbf{x} \in \mathbb{R}^n$ and allow a bounded perturbation of $+/-\epsilon$ to each element in $\mathbf{x}$. For each network, we take 100 random test inputs, filter out those that are incorrectly classified, apply an ϵ bounded perturbation

Table 1. Comparing certified accuracy and run time of LinSyn and AutoLiRPA.

Network Architecture		AutoLiRPA [47]		Our Method (new)	
		% certified	time (s)	% certified	time(s)
MNIST	4-Layer CNN with Swish	0.34	15	0.76	796
	4-Layer CNN with Gelu	0.01	359	0.72	814
	4-Layer CNN with Log Log	0.00	38	0.24	867
CIFAR	5-Layer CNN with Swish	0.03	69	0.35	1,077
	5-Layer CNN with Gelu	0.00	1,217	0.31	1,163
	5-Layer CNN with Log Log	0.59	98	0.69	717
SST-2	LSTM with sig tanh	0.93	37	0.91	1,074
	LSTM with hard tanh	-	-	0.64	2300
	LSTM with log log	0.16	1,072	0.82	2,859

Table 2. Comparing certified accuracy and run time of LinSyn and POPQORN.

Network Architecture		POPQORN [25]		Our Method (new)	
		% certified	time (s)	% certified	time(s)
SST-2	LSTM with sig tanh	0.93	1517	0.90	1,074

to the correctly classified inputs, and then attempt to prove the classification remains correct. We choose ϵ values common in prior work. For MNIST networks, in particular, we choose $\epsilon = 8/255$. For CIFAR networks, we choose $\epsilon = 1/255$. For SST-2 networks, we choose $\epsilon = 0.04$, and we only apply it to the first word embedding in the input sequence.

5.2 Experimental Results

Our experiments were designed to answer the following two questions: (1) How do LinSyn's linear bounds compare with handcrafted bounds? (2) How does the runtime of LinSyn compare to state-of-the-art linear bounding techniques? To answer these questions, we compare the effectiveness of LinSyn's linear bounds with the state-of-the-art linear bounding technique implemented in AutoLiRPA. To the best of our knowledge this is the only tool that can handle the activation functions we use in our benchmarks. As another comparison point, we also compare with POPQORN, a state-of-the-art linear bounding technique for LSTM networks. POPQORN tackles the challenge of computing tight linear bounds for $sigmoid(x) \times tanh(y)$ and $x \times sigmoid(y)$ using an expensive gradient descent based approach, and thus makes a good comparison point for runtime and accuracy. Our experiments were conducted on a computer with an Intel 2.6 GHz i7-6700 8-core CPU and 32GB RAM. Both AutoLiRPA and LinSyn are engineered to bound individual neurons in parallel. We configure each method to use up to 6 threads.

Overall Comparison First, we compare the overall performance of our new method and the default linear bounding technique in AutoLiRPA. The results are shown in Table 1. Here, Columns 1-2 show the name of the dataset and the type of neural networks. Columns 3-4 show the results of the default AutoLiRPA, including the percentage of inputs certified and the analysis time in seconds. Similarly, Columns 5-6 show the results of our new method.

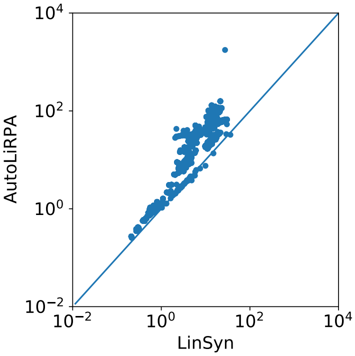


Fig. 7. Scatter plot comparing the final output interval width of LinSyn and AutoLiRPA.

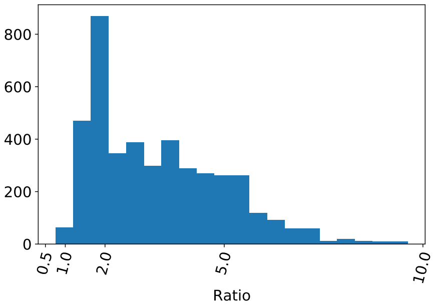


Fig. 8. Histogram of width ratios between AutoLiRPA and LinSyn. Ratio reported as $\frac{\text{AutoLiRPA}}{\text{LinSyn}}$.

The results in Table 1 show that, in terms of the analysis time, our method is slower, primarily due to the use of constraint solvers (namely dReal and the LP solver) but overall, the analysis speed is comparable to AutoLiRPA. However, in terms of accuracy, our method significantly outperforms AutoLiRPA. In almost all cases, our method was able to certify a much higher percentage of the inputs. For example, LinSyn more than quadruples the certified robustness of the *LSTM with log log* benchmark, and handles very well the relatively complex GeLU function. As for *SST-2: LSTM with hard tanh*, AutoLiRPA does not support the general $max(x, y)$ operation, so a comparison is not possible without significant engineering work.

The only exception to the improvement is *SST-2: LSTM with sig tanh*, for which the results are similar (.93 versus .91). In this case, there is likely little to be gained over the default, decomposition-based approach of AutoLiRPA in terms of tightness because the inputs to $sigmoid(x) \times tanh(y)$ and $x \times sigmoid(y)$ are not related, i.e., x and y are two separate variables. This is in contrast to, e.g., $swish(x) = x \times sigmoid(x)$, where the left-hand side and right-hand side of the multiplication *are* related.

In Table 2, we show a comparison between LinSyn and POPQORN. The result shows that our approach achieves similar certified robustness and runtime, even though POPQORN was designed to specifically target this particular type of LSTM architecture, while LinSyn is entirely generic.

Detailed Comparison Next, we perform a more in depth comparison of accuracy by comparing the widths of the final output neuron's intervals that are computed by AutoLiRPA and LinSyn. The results are shown in the scatter plot in Fig. 7 and the histogram in Fig. 8. Each point in the scatter plot represents a single output neuron x_i for a single verification problem. The x-axis is the width of the interval of the output neuron x_i (i.e. $u_i - l_i$) computed by LinSyn, and the y-axis is the width computed by AutoLiRPA. A point above the diagonal line indicates that LinSyn computed a tighter (smaller) final output interval. In the histogram, we further illustrate the accuracy gain as the width ratio, measured as $\frac{\text{AutoLiRPA}}{\text{LinSyn}}$. Overall, the results show that LinSyn is more accurate in nearly all cases, and LinSyn often produces final output bounds 2-5X tighter than AutoLiRPA.

6 Related Work

Linear Bound-based Neural Network Verification There is a large body of work on using linear-bounding techniques [36, 48, 34, 6, 45, 29, 30, 46, 27] and other abstract domains such as concrete intervals, symbolic intervals [44], and Zonotopes [15], for the purpose of neural network verification. All of these can be thought of as leveraging restricted versions of the polyhedral abstract domain [10, 9]. To the best of our knowledge, these approaches are the most scalable (in terms of network size) due to the use of approximations, but this also means they are less accurate than exact approaches. In addition, all these approaches have the limitation that they depend on bounds that are hand-crafted by an expert.

SMT solver-based Neural Network Verification There is also a large body of work on using exact constraint solving for neural network verification. Early works include solvers specifically designed for neural networks, such as Reluplex and Marabou [23, 24] and others [11], and leveraging existing solvers [12, 20, 5, 20, 4, 40, 19]. While more accurate, the reliance on an SMT solver typically limits their scalability. More recent work often uses solvers to refine the bounds computed by linear bounding [35, 37, 43, 42, 41]. Since the solvers leveraged in these approaches usually involve linear constraint solving techniques, they are usually only applicable to piece-wise linear activation functions such as ReLU and Max/Min-pooling.

7 Conclusions

We have presented LinSyn, a method for synthesizing linear bounds for arbitrary activation functions. The key advantage of LinSyn is that it can handle complex activation functions, such as Swish, GELU, and Log Log as a whole, allowing it to synthesize much tighter linear bounds than existing tools. Our experimental results show this increased tightness leads to drastically increased certified robustness, and tighter final output bounds.

References

1. Eran. https://github.com/eth-sri/eran (2021)
2. Alzantot, M., Sharma, Y., Elgohary, A., Ho, B.J., Srivastava, M., Chang, K.W.: Generating natural language adversarial examples. arXiv preprint arXiv:1804.07998 (2018)
3. Balunović, M., Baader, M., Singh, G., Gehr, T., Vechev, M.: Certifying geometric robustness of neural networks. Advances in Neural Information Processing Systems 32 (2019)
4. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1249–1264 (2019)
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A.V., Criminisi, A.: Measuring neural net robustness with constraints. In: Annual Conference on Neural Information Processing Systems. pp. 2613–2621 (2016)
6. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 3240–3247 (2019)
7. Byrd, R.H., Lu, P., Nocedal, J., Zhu, C.: A limited memory algorithm for bound constrained optimization. SIAM Journal on scientific computing **16**(5), 1190–1208 (1995)
8. Chabert, G., Jaulin, L.: Contractor programming. Artificial Intelligence **173**(11), 1079–1100 (2009)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252 (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 84–96 (1978)
11. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: International Conference on Uncertainty in Artificial Intelligence. pp. 550–559 (2018)
12. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. pp. 269–286 (2017)
13. Engstrom, L., Tran, B., Tsipras, D., Schmidt, L., Madry, A.: Exploring the landscape of spatial robustness. In: International Conference on Machine Learning. pp. 1802–1811. PMLR (2019)
14. Gao, S., Kong, S., Clarke, E.M.: dreal: An smt solver for nonlinear theories over the reals. In: International conference on automated deduction. pp. 208–214. Springer (2013)
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy. pp. 3–18 (2018)
16. Gomes, G.S.d.S., Ludermir, T.B.: Complementary log-log and probit: activation functions implemented in artificial neural networks. In: 2008 Eighth International Conference on Hybrid Intelligent Systems. pp. 939–942. IEEE (2008)
17. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021), https://www.gurobi.com

18. Hendrycks, D., Gimpel, K.: Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415 (2016)
19. Hu, H., Fazlyab, M., Morari, M., Pappas, G.J.: Reach-sdp: Reachability analysis of closed-loop systems with neural network controllers via semidefinite programming. In: 2020 59th IEEE Conference on Decision and Control (CDC). pp. 5929–5934. IEEE (2020)
20. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: International Conference on Computer Aided Verification. pp. 3–29 (2017)
21. Inc., W.R.: Mathematica, Version 12.3.1, https://www.wolfram.com/mathematica, champaign, IL, 2021
22. Kanbak, C., Moosavi-Dezfooli, S.M., Frossard, P.: Geometric robustness of deep networks: analysis and improvement. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4441–4449 (2018)
23. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117 (2017)
24. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.W.: The Marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452 (2019)
25. Ko, C.Y., Lyu, Z., Weng, L., Daniel, L., Wong, N., Lin, D.: Popqorn: Quantifying robustness of recurrent neural networks. In: International Conference on Machine Learning. pp. 3468–3477. PMLR (2019)
26. Kong, S., Solar-Lezama, A., Gao, S.: Delta-decision procedures for exists-forall problems over the reals. In: International Conference on Computer Aided Verification. pp. 219–235. Springer (2018)
27. Mohammadinejad, S., Paulsen, B., Wang, C., Deshmukh, J.V.: Diffrnn: Differential verification of recurrent neural networks. arXiv preprint arXiv:2007.10135 (2020)
28. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to interval analysis, vol. 110. Siam (2009)
29. Paulsen, B., Wang, J., Wang, C.: Reludiff: Differential verification of deep neural networks. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). pp. 714–726. IEEE (2020)
30. Paulsen, B., Wang, J., Wang, J., Wang, C.: Neurodiff: scalable differential verification of neural networks using fine-grained approximation. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 784–796. IEEE (2020)
31. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I.: Improving language understanding by generative pre-training (2018)
32. Ramachandran, P., Zoph, B., Le, Q.V.: Searching for activation functions. arXiv preprint arXiv:1710.05941 (2017)
33. Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A., Vechev, M.: Scalable polyhedral verification of recurrent neural networks. In: International Conference on Computer Aided Verification. pp. 225–248. Springer (2021)
34. Shi, Z., Zhang, H., Chang, K.W., Huang, M., Hsieh, C.J.: Robustness verification for transformers. International Conference on Learning Representations (2020)
35. Singh, G., Ganvir, R., Pschel, M., Vechev, M.: Beyond the single neuron convex barrier for neural network certification. In: Advances in Neural Information Processing Systems (NeurIPS) (2019)
36. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp. 41:1–41:30 (2019)

37. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (2019)
38. Singla, V., Singla, S., Feizi, S., Jacobs, D.: Low curvature activations reduce overfitting in adversarial training. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 16423–16433 (2021)
39. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
40. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. International Conference on Learning Representations (2019)
41. Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. In: International Conference on Computer Aided Verification. pp. 18–42. Springer (2020)
42. Tran, H.D., Lopez, D.M., Musau, P., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Star-based reachability analysis of deep neural networks. In: International Symposium on Formal Methods. pp. 670–686. Springer (2019)
43. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Annual Conference on Neural Information Processing Systems. pp. 6369–6379 (2018)
44. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security Symposium. pp. 1599–1614 (2018)
45. Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: International Conference on Machine Learning. pp. 5273–5282 (2018)
46. Wu, Y., Zhang, M.: Tightening robustness verification of convolutional neural networks with fine-grained linear approximation. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 11674–11681 (2021)
47. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.W., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.J.: Automatic perturbation analysis for scalable certified robustness and beyond. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1129–1141. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper/2020/file/0cbc5671ae26f67871cb914d81ef8fc1-Paper.pdf
48. Zhang, H., Weng, T.W., Chen, P.Y., Hsieh, C.J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in neural information processing systems. pp. 4939–4948 (2018)

Short papers

Kmclib: Automated Inference and Verification of Session Types from OCaml Programs

Keigo Imai[1], Julien Lange[2], and Rumyana Neykova[3]

[1] Gifu University, Gifu, Japan, keigoi@gifu-u.ac.jp
[2] Royal Holloway, University of London, UK, julien.lange@rhul.ac.uk
[3] Brunel University London, UK, rumyana.neykova@brunel.ac.uk

Abstract. Theories and tools based on multiparty session types offer correctness guarantees for concurrent programs that communicate using message-passing. These guarantees usually come at the cost of an intrinsically top-down approach, which requires the communication behaviour of the entire program to be specified as a global type.
This paper introduces `kmclib`: an OCaml library that supports the development of *correct* message-passing programs without having to write any types. The library utilises the meta-programming facilities of OCaml to automatically infer the session types of concurrent programs and verify their compatibility (k-MC [15]). Well-typed programs, written with `kmclib`, do not lead to communication errors and cannot get stuck.

Keywords: Multiparty Session Types · Concurrent Programming · OCaml

1 Introduction

Multiparty session types (MPST) [5] are a popular type-driven technique to ensure the correctness of concurrent programs that communicate using message-passing. The key benefit of MPST is to guarantee statically that the components of a program have compatible behaviours, and thus no components can get permanently stuck. Many implementations of MPST in different programming languages have been proposed in the last decade [2,4,6,10,12,16–18,20,23], however, all suffer from a notable shortcoming: they require programmers to adopt a top-down approach that does not fit well in modern development practices. When changes are frequent and continual (e.g., continuous delivery), re-designing the program and its specification at every change is not feasible.

Most MPST theories and tools advocate an intrinsically top-down approach. They require programmers to specify the communication (often in the form of a global type) of their programs before they can be type-checked. In practice, type-checking programs against session types is very difficult. To circumvent the problem, most implementations of MPST rely on *external* toolings that generate code from a global type, see e.g., all works based on the Scribble toolchain [22].

In this paper, we present an OCaml library, called `kmclib` [8,9], which supports the development of programs that enjoy all the benefits of MPST while avoiding their main drawbacks. The `kmclib` library guarantees that threads in

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 379–386, 2022.
https://doi.org/10.1007/978-3-030-99524-9_20

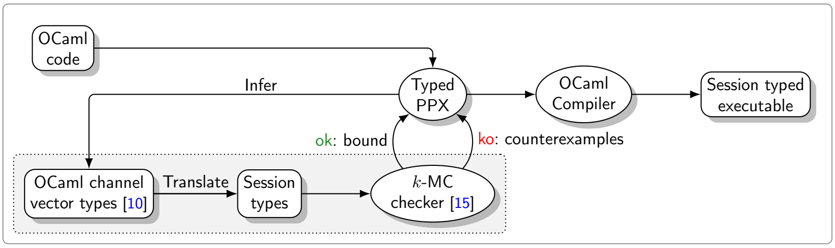


Fig. 1: Workflow of the `kmclib` library (the PPX plugin is the shaded box).

well-typed programs will not get stuck. The library also enables *bottom-up development*: programmers write message-passing programs in a natural way, without having to write session types. Our library is built on top of *Multicore OCaml* [21] that offers highly scalable and efficient concurrent programming, but does not provide any static guarantees wrt. concurrency.

Figure 1 gives an overview of `kmclib`. Its implementation combines the power of the *type-aware* macro system of OCaml (Typed PPX) with two recent advances in the session types area: an encoding of MPST in OCaml (channel vector types [10]) and a session type compatibility checker (k-MC checker [15]). To our knowledge, this is the first implementation of type inference for MPST and the first integration of compatibility checking in a programming language.

The `kmclib` library [8,9] offers several advantages compared to earlier MPST implementations. **(1)** It is *flexible*: programmers can implement communication patterns (e.g., fire-and-forget patterns [15]) that are not expressible in the synchrony-oriented syntax of global types. **(2)** It is *lightweight* as it piggybacks on OCaml's type system to check and infer session types, hence lifting the burden of writing session types off the programmers. **(3)** It is *user-friendly* thanks to its integration in Visual Studio Code, where compatibility violations are mapped to precise locations in the code. **(4)** It is *well-integrated* into the natural edit-compile-run cycle. Although compatibility is checked by an external tool, this step is embedded as a compilation step and thus hidden from the user.

2 Safe Concurrent Programming in Multicore OCaml

We give an overview of the features and usage of `kmclib` using the program in Figure 2 (top) which calculates Fibonacci numbers. The program consists of three concurrent *threads* (`user`, `master`, and `worker`) that interact using point-to-point message-passing. Initially, the `user` thread sends a request to the `master` to start the calculation, then waits for the `master` to return a work-in-progress message, or the final result. After receiving the result, the `user` sends back a stop message. Upon receiving a new request, the `master` splits the initial computation in two and sends two tasks to a `worker`. For each task that the `worker` receives, it replies with a result. The `master` and `worker` threads are recursive and terminate only upon receiving a stop message.

```
let KMC (uch,mch,wch) = [%kmc.gen (u,m,w)]

let user () =
  let uch = send uch#m#compute 42 in
  let rec loop uch : unit =
    match receive uch#m with
    | `wip(res, uch) ->
      printf "in progress: %d\n" res;
      loop uch
    | `result(res, uch) ->
      printf "result: %d\n" res;
      send uch#m#stop ()
  in loop uch

let worker () =
  let rec loop wch : unit =
    match receive wch#m with
    | `task(num, wch) ->
      loop (send wch#m#result (fib num))
    | `stop((), wch) -> wch
  in loop wch
```

```
let master () =
  let rec loop (mch : [%kmc.check u]) : unit =
    match receive mch#u with
    | `compute(x, mch) ->
      let mch = send mch#w#task (x - 2) in
      let mch = send mch#w#task (x - 1) in
      let `result(r1, mch) = receive mch#w in
      let mch = send mch#u#wip r1 in
      let `result(r2, mch) = receive mch#w in
      loop (send mch#u#result (r1 + r2))
    | `stop((), mch) ->
      send mch#w#stop ()
  in loop mch

let () =
  let ut = Thread.create user () in
  let mt = Thread.create master () in
  let wt = Thread.create worker () in
  List.iter Thread.join [ut;mt;wt]
```

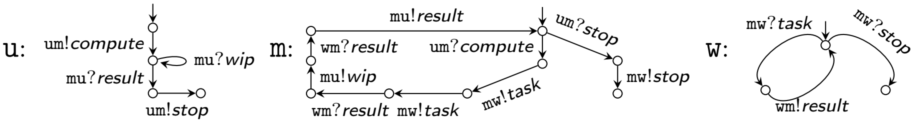
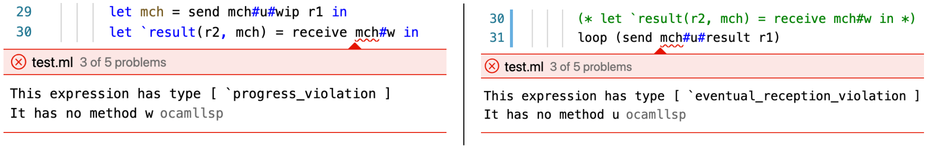


Fig. 2: Example of kmclib program (top) and *inferred* session types (bottom).

Figure 2 (bottom) gives a session type for each thread, i.e., the behaviour of each thread wrt. communication. For clarity we represent session types as a communicating finite state machines (CFSM [1]), where ! (resp. ?) denotes sending (resp. receiving). For example, um!*compute* means that the user is sending to the master a message *compute*, while um?*compute* says that the master receives *compute* from the user. Our library infers these CFSM representations from the OCaml code, in Figure 2 (top), and verifies *statically* that the three threads are *compatible*, hence no thread can get stuck due to communication errors. If compatibility cannot be guaranteed, the compiler reports the kind of violations (i.e., *progress* or *eventual reception* error) and their locations in the code. Figure 3 shows how such semantic errors are reported visually in Visual Studio Code.

Albeit simple, the common communication pattern used in Figure 2 cannot be expressed as a global type, and thus cannot be implemented in previous MPST implementations. Concretely, global types cannot express the intrinsically asynchronous interactions between the master and worker threads (i.e., the master may send a second task message, while the worker sends a result).

Programming with kmclib. To enable safe message-passing programs, kmclib provides two communication primitives, send and receive, and two primitives for channel creation (KMC and %kmc.gen). Our library supports all the features of traditional MPST implementations and have similar limitations (fixed number of participant, no delegation, etc). We only give a user-oriented description of these primitives here (see [7, §A] for an overview of their implementations).



```
let mch = send mch#u#wip r1 in
let `result(r2, mch) = receive mch#w in
```

test.ml 3 of 5 problems

```
This expression has type [ `progress_violation ]
It has no method w ocamllsp
```

```
(* let `result(r2, mch) = receive mch#w in *)
loop (send mch#u#result r1)
```

test.ml 3 of 5 problems

```
This expression has type [ `eventual_reception_violation ]
It has no method u ocamllsp
```

Fig. 3: Examples of type errors: progress (left) and eventual reception (right).

The crux of `kmclib` is the **session channel creation**: `[%kmc.gen (u,m,w)]` at Line 1. This primitive takes a tuple of *role names* as argument (i.e., `(u,m,w)`) and returns a tuple of communication channels, which are bound to `(uch,mch,wch)`. These channels will be used by the threads implementing roles `user` (Lines 3-13), `worker` (Lines 15-21), and `master` (Lines 22-34). Channels are implemented using concurrent queues from Multicore OCaml (`Domainslib.Chan.t`) but other underlying transports can easily be provided.

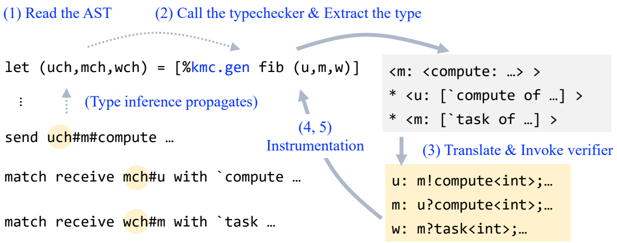
Threads send and receive messages over these channels using the communication primitives provided by `kmclib`. The send primitive requires three *arguments*: a channel, a destination role, and a message. For instance, the `user` sends a request to the `master` with `send uch#m#compute 20` where `uch` is the user's communication channel, `m` indicates the destination, and `compute 20` is the message (consisting of a label and a payload). Observe that a sending operation returns a new channel which is to be used in the continuation of the interactions, e.g., `uch` bound at Line 4, which must be used linearly (see [7] for details). Receiving messages works in a similar way to sending messages, e.g., see Line 6 where the `user` waits for a message from the `master` with `receive uch#m`. We use OCaml's pattern matching to match messages against their labels and bind the payload and continuation channel. See, e.g., Lines 7-10 where the `user` expects either a `wip` or `result` message. The receive primitive returns the payload `res` and a new communication channel `uch`.

New thread instances are spawned in the usual way; see Lines 36-39. The code at Line 40 waits for them to terminate.

Compatibility and error reporting. While the code in Figure 2 may appear unremarkable, it hides a substantial machinery that guarantees that, if a program type-checks, then its constituent threads are safe, i.e., no thread gets permanently stuck and all messages that are sent are eventually received. This property is ensured by `kmclib` using OCaml's type inference and PPX plugins to infer a session type from each thread then check whether these session types are *k-multiparty compatible* (k-MC) [15].

If a system of session types is k-MC, then it is safe [15, Theorem 1], i.e., it has the *progress* property (no role gets permanently stuck in a receiving state) and the *eventual reception* property (all sent messages are eventually received). Checking k-MC notably involves checking that all their executions (where each channel contains at most k messages) satisfy progress and eventual reception.

The k-MC-checker [15] performs a bounded verification to discover the *least* k for which a system is k-MC, up-to a specified upper bound N. In the `kmclib`


(1) Read the AST
(2) Call the typechecker & Extract the type
(Type inference propagates)
(4, 5)
Instrumentation
(3) Translate & Invoke verifier

```
let (uch,mch,wch) = [%kmc.gen fib (u,m,w)]
  ⋮
send uch#m#compute …
match receive mch#u with `compute …
match receive wch#m with `task …
```

```
<m: <compute: …> >
* <u: [`compute of …] >
* <m: [`task of …] >
```

```
u: m!compute<int>;…
m: u?compute<int>;…
w: m?task<int>;…
```

Fig. 4: Inference of session types from OCaml code.

API, this bound can be optionally specified with [%kmclib.gen *roles* ~bound:*N*]. The k-MC-checker emits an error if the bound is insufficient to guarantee safety.

The [%kmc.gen (u,m,w)] primitive also feeds the results of k-MC checking back to the code. If the inferred session types are k-MC, then channels for roles u, m and w can be generated, otherwise a type error is raised. We have modified the k-MC-checker to return counterexample traces when the verification fails. This helps give actionable feedback to the programmer, as counterexample traces are translated to OCaml types and inserted at the hole corresponding to [%kmc.gen]. This has the effect of reporting the precise location of the errors.

To report errors in a function parameter, we provide an *optional* macro for types: [%kmc.check rolename] (see faded code in Line 23). Figure 3 shows examples of such error reports. The left-hand-side shows the reported error when Line 26 is commented out, i.e., the master sends one task, but expects two result messages; hence progress is violated since the master gets stuck at Line 30. The right-hand-side shows the reported error when Line 30 is commented out. In this case, variable mch in Line 31 (master) is highlighted because the master fails to consume a message from channel mch.

3 Inference of Session Types in kmclib

The kmclib API. The kmclib primitives allow the vanilla OCaml typechecker to infer the session structure of a program, while simultaneously providing a user-friendly communication API for the programmer. To enable inference of session types from concurrent programs, we leverage OCaml's structural typing and row polymorphism. In particular, we reuse the encoding from [10] where input and output session types are encoded as polymorphic variants and objects in OCaml. In contrast to [10] which relies on programmers writing global types prior to type-checking, kmclib infers and verifies local session types automatically, without requiring any additional type or annotation.

Typed PPX Rewriter. To extract and verify session types from a piece of OCaml code, the kmclib library makes use of OCaml PreProcessor eXtensions (PPX) plugins which provide a powerful meta-programming facility. PPX plugins are invoked during the compilation process to manipulate or translate the abstract syntax tree (AST) of the program. This is often used to insert additional definitions, e.g., pretty-printers, at compile-time.

A key novelty of `kmclib` is the combination of PPX with a form of *type-aware translation*, whereas most PPX plugins typically perform purely syntactic (type-unaware) translations. Figure 4 shows the workflow of the PPX rewriter, overlayed on code snippets from Figure 2. The inference works as follows.

(1) The plugin reads the AST of the program code to replace the `[%kmc.gen]` primitive with a *hole*, which can have *any* type.
(2) The plugin invokes the *typechecker* to get the *typed* AST of the program. In this way, the type of the hole is *inferred* to be a tuple of *channel object types* whose structure is derived from their usages (i.e., `mch#u#compute`).
To enable this propagation, we introduce the idiom "`let (KMC ...) = ...`" which enforces the type of the hole to be *monomorphic*. Otherwise, the type would be too general and this would spoil the type propagation, see [7, § B].
(3) The inferred type is translated to a system of (local) *session types*, which are passed to the k-MC-checker.
(4) If the system is k-MC, then it is safe and the plugin *instruments* the code to allocate a fresh channel tuple (i.e., concurrent queues) at the hole.
(5) Otherwise, the k-MC-checker returns a *violation trace* which is translated back to an OCaml type and inserted at the hole, to report a more precise error.

The translation is limited inside the `[%kmc.gen]` expression, retaining a clear correspondence between the original and translated code. It can be understood as a form of *ad hoc polymorphism* reminiscent of type classes in Haskell. Like the Haskell typechecker verifies whether a type belongs to a class or not, the `kmclib` verifies whether the set of session types belongs to the class of k-MC systems.

4 Conclusion

We have developed a practical library for safe message-passing programming. The library enables developers to program and verify arbitrary communication patterns without the need for type annotations or user-operated external tools. Our *automated verification* approach can be applied to other general-purpose programming languages. Indeed it mainly relies on two ingredients: static structural typing and metaprogramming facilities. Both are available, with a varying degree of support, in, e.g., Scala, Haskell, TypeScript, and F#.

Our work is reminiscent of automated software model checking which has a long history (see [11] for a survey). There are few works on inference and verification of behavioural types, i.e., [3, 13, 14, 19]. However, Perera et al. [19] only present a prototype research language, while Lange et al. [3, 13, 14] propose verification procedures for Go programs that rely on external tools which are not integrated with the language nor its type system. To our knowledge, ours is the first implementation of type inference for MPST and the first integration of session types compatibility checking within a programming language.

Acknowledgements. This work is partially supported by KAKENHI 17K12662, 21K11827, 21H03415, and EPSRC EP/W007762/1.

References

1. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
2. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in Go: statically-typed endpoint apis for dynamically-instantiated communication structures. PACMPL **3**(POPL), 29:1–29:30 (2019), https://dl.acm.org/citation.cfm?id=3290342
3. Dilley, N., Lange, J.: Automated Verification of Go Programs via Bounded Model Checking. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. pp. 1016–1027. IEEE (2021). https://doi.org/10.1109/ASE51524.2021.9678571
4. Harvey, P., Fowler, S., Dardha, O., J. Gay, S.: Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming (ECOOP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, p. 30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.12
5. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008. pp. 273–284 (2008). https://doi.org/10.1145/1328438.1328472
6. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE 2016. pp. 401–418 (2016). https://doi.org/10.1007/978-3-662-49665-7_24
7. Imai, K., Lange, J., Neykova, R.: kmclib: Automated inference and verification of session types (extended version). CoRR **abs/2111.12147** (2021), https://arxiv.org/abs/2111.12147
8. Imai, K., Lange, J., Neykova, R.: kmclib: A communication library with static guarantee on concurrency (2022), https://github.com/keigoi/kmclib
9. Imai, K., Lange, J., Neykova, R.: Kmclib: Artifact for the TACAS 2022 paper (2022). https://doi.org/10.5281/zenodo.5887544
10. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference). LIPIcs, vol. 166, pp. 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.9
11. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4) (Oct 2009). https://doi.org/10.1145/1592434.1592438
12. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: PPDP 2016. pp. 146–159 (2016). https://doi.org/10.1145/2967973.2968595
13. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: liveness and safety for channel-based programming. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 748–761. ACM (2017). https://doi.org/10.1145/3009837.3009847
14. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1137–1148. ACM (2018). https://doi.org/10.1145/3180155.3180157

15. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 97–117. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_6
16. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Generating Interactive WebSocket Applications in TypeScript. Electronic Proceedings in Theoretical Computer Science **314**, 12–22 (Apr 2020). https://doi.org/10.4204/EPTCS.314.2
17. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F♯. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138. ACM (2018). https://doi.org/10.1145/3178372.3179495
18. Ng, N., de Figueiredo Coutinho, J.G., Yoshida, N.: Protocols by default - safe MPI code generation based on session types. In: CC 2015. pp. 212–232 (2015). https://doi.org/10.1007/978-3-662-46663-6_11
19. Perera, R., Lange, J., Gay, S.J.: Multiparty compatibility for concurrent objects. In: PLACES 2016. pp. 73–82 (2016). https://doi.org/10.4204/EPTCS.211.8
20. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP 2017. pp. 24:1–24:31 (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
21. Sivaramakrishnan, K.C., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto ocaml. Proc. ACM Program. Lang. **4**(ICFP), 113:1–113:30 (2020). https://doi.org/10.1145/3408995
22. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble protocol language. In: Abadi, M., Lluch-Lafuente, A. (eds.) Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8358, pp. 22–41. Springer (2013). https://doi.org/10.1007/978-3-319-05119-2_3
23. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 148:1–148:30 (2020). https://doi.org/10.1145/3428216

Automated Translation of Natural Language Requirements to Runtime Monitors

Ivan Perez[1] (✉), Anastasia Mavridou[2] (✉), Tom Pressburger[3], Alwyn Goodloe[4], and Dimitra Giannakopoulou[3]*

[1] National Institute of Aerospace, Hampton, Virginia
[2] KBR Inc., NASA Ames Research Center, Moffett Field, California
[3] NASA Ames Research Center, Moffett Field, California
[4] NASA Langley Research Center, Hampton, Virginia
{ivan.perezdominguez,anastasia.mavridou}@nasa.gov

Abstract. Runtime verification (RV) enables monitoring systems at runtime, to detect property violations early and limit their potential consequences. This paper presents an end-to-end framework to capture requirements in structured natural language and generate monitors that capture their semantics faithfully. We leverage NASA's Formal Requirement Elicitation Tool (FRET), and the RV system Copilot. We extend FRET with mechanisms to capture additional information needed to generate monitors, and introduce Ogma, a new tool to bridge the gap between FRET and Copilot. With this framework, users can write requirements in an intuitive format and obtain real-time C monitors suitable for use in embedded systems. Our toolchain is available as open source.

1 Introduction

Safety-critical systems, such as aircraft, automobiles, and power systems, where failure can result in injury or death of a human [23], must undergo extensive assurance. The verification process must ensure that the system satisfies its requirements under realistic operating conditions and that there is no unintended behavior. Verification rests on possessing a precise statement of requirements, arguably one of the most difficult tasks in engineering reliable software.

Runtime verification (RV) [21, 19, 5] has the potential to enable the safe operation of complex safety-critical systems. RV monitors can be used to detect and respond to property violations during missions, as well as to verify implementations and simulations at design time. For monitors to be effective, they must faithfully reflect the mission requirements, which is difficult for non-trivial systems because correctness properties must be expressed in a precise mathematical formalism while requirements are generally written in natural language.

The focus of this paper is to provide an end-to-end framework that takes as input requirements and other necessary data and provides mechanisms to 1) help the user deeply understand the semantics of these requirements, 2) automatically generate formalizations and 3) produce RV monitors that faithfully

[*] Author contributed to this work prior to joining AWS.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 387–395, 2022.
https://doi.org/10.1007/978-3-030-99524-9_21

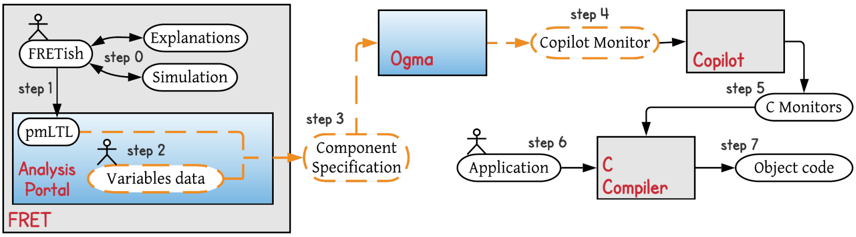


Fig. 1: Step-by-step workflow

capture the semantics of the requirements. We leverage NASA's Formal Requirement Elicitation Tool (FRET) [17, 18] and the runtime monitoring system Copilot [29, 36, 35]. FRET allows users to express and understand requirements through its intuitive structured natural language (named FRETISH) and elicitation mechanisms, and generates formalizations in temporal logic. Copilot allows users to specify monitors and compile them to hard real-time C code.

The contribution of this paper is the tight integration of the FRET-Copilot tools to support the automated synthesis of executable RV monitors directly from requirement specifications. In particular, we present:

- A new tool, named Ogma, that receives requirement formalizations and variable data from FRET and compiles these into Copilot monitors.
- An extension of the FRET analysis portal to support the generation and export of specifications that can be directly digested by Ogma.
- Preliminary experimental results that evaluate the proposed workflow.

All tools needed by our workflow are available as open source [2, 1, 4].

Related Work. A number of runtime verification languages and systems have been applied in resource-constrained environments [39, 13, 6, 7, 37, 28]. In contrast to our work, these systems do not provide a direct translation from natural language. Several tools [25, 14, 16, 24, 8] formalize natural-language like requirements, but not for the purpose of generating runtime monitors. The STIMULUS tool [22] allows users to express requirements in an extensible, natural-like language that is syntactic sugar for hierarchical state machines. The machines then act as monitors that can be used to validate requirements during the design and testing phases, but are not intended to be used at runtime. FLEA [10] is a formal language for expressing requirements that compiles to runtime monitors in a garbage collected language, making it harder to use in embedded systems; in contrast, our approach generates hard real-time code.

2 Step-by-step Framework Workflow

To integrate FRET and Copilot, we extended the FRET analysis portal and created the Ogma tool. Figure 1 shows the step-by-step workflow of the complete framework - dashed lines represent the newly added steps (2, 3, and 4). Once requirements are written in FRETISH, FRET helps users understand and refine their requirements through various explanations and simulation (step 0). Next, NL: “*While flying, if the airspeed is below 100 m/s, the autopilot shall increase the airspeed to at least 100 m/s within 10 seconds.*”

FRETish: `in flight mode if airspeed < 100 the aircraft shall within 10 seconds satisfy (airspeed >= 100)`

pmLTL: H (Lin_flight→(Y (((O$_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O$_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) S (((O$_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O$_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) & Fin_flight)))) & ((!Lin_flight) S ((!Lin_flight) & Fin_flight)) → (((O$_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O$_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) S (((O$_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → (O$_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) & Fin_flight)),

where `Fin_flight` (First timepoint in flight mode) is `flight & (FTP | Y !flight)`, `Lin_flight` (Last timepoint in flight mode) is `!flight & Y flight`, `FTP` (First Time Point) is `! Y true`.

Fig. 2: Running example in Natural Language (NL), FRETISH, and pmLTL forms.

FRET automatically translates requirements (step 1) into pure Past-time Metric Linear Temporal Logic (pmLTL) formulas. Next, information about the variables referenced in the requirements must be provided by the user (step 2). The formulas, as well as the provided variables' data, are then combined to generate the Component Specification (step 3). Based on this specification, OGMA creates a complete COPILOT monitor specification (step 4). COPILOT then generates the C Monitor (step 5), which is given along with other C code (step 6) to a C Compiler for the generation (step 7) of the final object code.

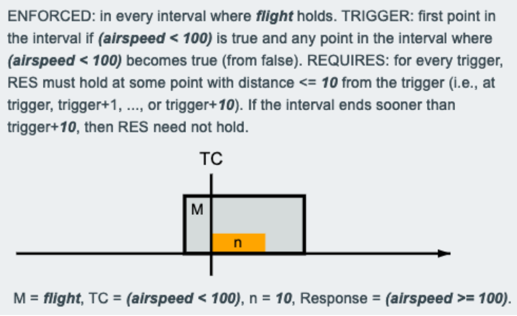
Running Example. The next sections illustrate each workflow step using a flight-critical system requirement: airplanes should always avoid stalling (a stall is a sudden loss of lift, which may lead to a loss of control). To avoid stalls, they should fly above a certain speed, known as *stall speed* (as well as stay below a critical angle of attack). Our running requirement example is captured in natural language in Figure 2. For the purposes of this example, we consider the airspeed threshold to be 100 m/s and the correction time to be 10 seconds.

3 FRET Steps

Next we discuss FRET, the requirements tool that constitutes our frontend.

Step 0:* fretish *and semantic nuances. A FRETISH requirement (see running example in Figure 2) contains up to six fields: `scope`, `condition`, `component*`, `shall*`, `timing`, and `response*`. Fields marked with * are mandatory.

`component` specifies the component that the requirement refers to (e.g., aircraft). `shall` expresses that the component's behavior must conform to the requirement. `response` is of the form *satisfy R*, where *R* is a Boolean condition (e.g., satisfy airspeed ≥ 100). `scope` specifies the period when the requirement holds during the execution of the system, e.g., when “in flight mode”. `condition` is a Boolean expression that further constrains when the `response` shall occur (e.g., the requirement becomes relevant only upon airspeed ≤ 100 becoming true). `timing` specifies when the `response` must occur (e.g., within 10 seconds).

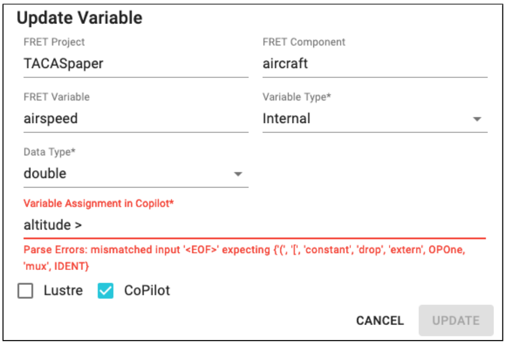


ENFORCED: in every interval where ***flight*** holds. TRIGGER: first point in the interval if ***(airspeed < 100)*** is true and any point in the interval where ***(airspeed < 100)*** becomes true (from false). REQUIRES: for every trigger, RES must hold at some point with distance <= ***10*** from the trigger (i.e., at trigger, trigger+1, ..., or trigger+***10***). If the interval ends sooner than trigger+***10***, then RES need not hold.

M = ***flight***, TC = ***(airspeed < 100)***, n = ***10***, Response = ***(airspeed >= 100)***.

Fig. 3: FRET explanations

Getting a temporal requirement right is usually a tricky task since such requirements are often riddled with semantic subtleties. To help the user, FRET provides a simulator and semantic explanations [17]. For example, the diagram in Figure 3 explains that the requirement is only relevant within the grayed box M (while in flight mode). TC represents the triggering condition (airspeed $<$ 100) and the orange band, with a duration of n=10 seconds, states that the response (airspeed >= 100) is required to hold at least once within the 10 seconds duration, assuming that flight mode holds for at least 10 seconds.



Update Variable

FRET Project: TACASpaper

FRET Component: aircraft

FRET Variable: airspeed

Variable Type*: Internal

Data Type*: double

Variable Assignment in Copilot*: altitude >

Parse Errors: mismatched input '<EOF>' expecting {'(', '[', 'constant', 'drop', 'extern', OPOne, 'mux', IDENT}

□ Lustre ☑ CoPilot

Fig. 4: FRET variable editor

Step 1:* fretish *to pmLTL. For each FRETISH requirement, FRET generates formulas in a variety of formalisms. For the COPILOT integration, we use the generated pmLTL formulas (Figure 2) Clearly, manually writing such formulas can be quite error-prone, while the FRET formalization process has been extensively tested through its *formalization verifier* [17].

Steps 2 & 3: Variables data and Component Specification. We extended FRET's analysis portal [3] to capture the information needed to generate Component Specifications for OGMA. To generate a specification, the user must indicate the type (i.e., input, output, internal) and data type (integer, Boolean, double, etc) of each variable (Figure 4).Internal variables represent expressions of input and output variables; if the same expression is used in multiple requirements, an internal variable can be used to substitute it and simplify the requirements. The user must *assign* an expression to each internal variable. In our example, the `flight` internal variable is defined by the expression `altitude > 0.0`, where `altitude` is an input variable. Internal variable assignments can be defined in Lustre [20] or Copilot [29]. Integrated Lustre and Copilot parsers identify parsing errors and return feedback (Figure 4). Once steps 1 and 2 are completed, FRET generates a Component Specification, which contains all requirements in pmLTL and Lustre code, as well as variable data that belong to the same system component.

4 Ogma Steps

Ogma is a command-line tool to produce monitoring applications. Ogma generates monitors in Copilot, and also supports integrating them into larger systems, such as applications built with NASA's core Flight System (cFS) [40].

Step 4: Copilot Monitors. Ogma provides a command `fret-component-spec` to process Component Specifications. The command traverses the Abstract Syntax Tree of the Component Specification, and converts each tree node into its Copilot counterpart. Input and output variables in FRET become *extern* streams in Copilot, or time-varying sources of information needed by the monitors:

```
airspeed :: Stream Double
airspeed = extern "airspeed" Nothing
```

Internal variables are also mapped to streams. Each requirement's pmLTL formula is translated into a Boolean stream, paired with a C handler *triggered* when the requirement is violated. In the example below, the property we monitor is associated with a handler, `handlerpropAvoidStall`, which must be implemented separately in C by the user to determine how to address property violations:

```
propAvoidStall :: Stream Bool
propAvoidStall = ((PTLTL.alwaysBeen ((((not (flight)) && ... )))))
spec = trigger "handlerpropAvoidStall" (not propAvoidStall) []
```

5 Copilot Steps

Copilot is a stream-based runtime monitoring language. Copilot streams may contain data of different types. At the top level, specifications consist of pairs of Boolean streams, together with a C handler to be called when the current sample of a stream becomes true. For a detailed introduction to Copilot, see [29].

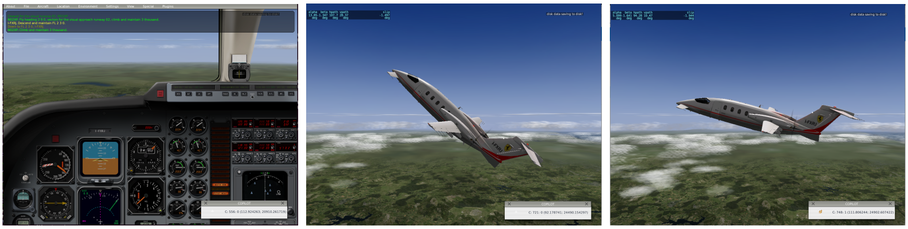
Step 5: C Monitors. Ogma generates self-contained Copilot monitoring specifications, which can be further compiled into C99 by just compiling and running the Copilot specifications with a Haskell compiler. This process produces two files: a C header and a C implementation.

Step 6: Larger Applications. The C files generated by Copilot are designed to be integrated into larger applications. They provide three connection endpoints: extern variables, a `step` function, and handler functions, which users implement to handle property violations. The code generated has no dynamic memory allocation, loops or recursive calls, it executes in predictable memory and time. For our running example, the header file generated by Copilot declares:

```
extern bool flight;                 extern float airspeed;
void handlerpropAvoidStall(void);   void step(void);
```

Commonly, the calling application will poll sensors, write their values to global variables, call the `step` function, and implement handlers that log property violations or execute corrective actions. Users are responsible for compiling and linking the Copilot code together with their application (step 7).

We also used the running requirement in this paper to monitor a flight in the simulator X-Plane. We wrote an X-Plane plugin to show the state of the C



(a) Cruising (b) Stall (c) Recovery

Fig. 5: Demonstration of Copilot monitor running as X-Plane plugin.

monitor and some additional information on the screen (Fig. 5a). To test the code, we brought an aircraft to a stall by increasing the angle of attack, which also lowered the airspeed (Fig. 5b). After 10 seconds below the specified threshold, the monitor became active, remaining on after executing a stall recovery (Fig. 5c).

6 Preliminary Results

We report on experiments with monitors generated from the publicly available Lockheed Martin Cyber-Physical System (LMCPS) challenge problems [11, 12], which are a set of industrial Simulink model benchmarks and natural language requirements developed by domain experts. LMCPS requirements were previously written in fretish [27, 26] by a subset of the authors and were analyzed against the provided models using model checking.

In this paper, we reuse the fretish requirements to generate monitors and compare our runtime verification results with the model checking results of [26]. For each Simulink model we generated C code through the automatic code generation feature of Matlab/Simulink. We then attached the generated C monitors to the C code and used the property-based testing system QuickCheck [9] to generate random streams of data, feed them to the system under observation, and report if any of the monitors were activated, based on [30, 31, 34].

We experimented with the Finite State Machine (FSM) and the Control Loop Regulators (REG) LMCPS challenges. For both challenges, our results are consistent with the model checking results - QuickCheck found inputs that activated the monitors, indicating that some requirements were not satisfied. Moreover, it returned results within seconds in cases where model checkers timed out. See [33] for details on the results and [32] for a reproducible artifact.

7 Conclusion

We described an end-to-end framework in which requirements written in structured natural language can be equivalently transformed into monitors and be analyzed against C code. Our framework ensures that requirements and analysis activities are fully aligned: C monitors are derived directly from requirements and not handcrafted. The design of our toolchain facilitates extension with additional front-ends (e.g., JKind Lustre [15]), and backends (e.g., R2U2 [38]). In the future, we plan to explore more use cases, including some from real drone test flights.

References

1. Copilot. `https://github.com/Copilot-Language/copilot/`. Accessed Oct 04, 2021.
2. FRET: Formal Requirements Elicitation Tool. `https://github.com/NASA-SW-VnV/fret/`. Accessed Oct 04, 2021.
3. FRET: Formal Requirements Elicitation Tool - User Manual. `https://github.com/NASA-SW-VnV/fret/blob/master/fret-electron/docs/_media/userManual.md`. See Section "Exporting for Analysis". Accessed Oct 04, 2021.
4. Ogma. `https://github.com/nasa/ogma/`. Accessed Oct 04, 2021.
5. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.
6. J. Baumeister, B. Finkbeiner, S. Schirmer, M. Schwenger, and C. Torens. RT-Lola cleared for take-off: Monitoring autonomous aircraft. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification*, pages 28–39, Cham, 2020. Springer International Publishing.
7. S. Biewer, B. Finkbeiner, H. Hermanns, M. A. Köhl, Y. Schnitzer, and M. Schwenger. RTLola on board: Testing real driving emissions on your phone. In J. F. Groote and K. G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–372, Cham, 2021. Springer International Publishing.
8. A. Boteanu, T. Howard, J. Arkin, and H. Kress-Gazit. A model for verifiable grounding and execution of complex natural language instructions. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2649–2654, 2016.
9. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM Sigplan Notices*, 46(4):53–64, 2011.
10. D. Cohen, M. S. Feather, K. Narayanaswamy, and S. S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering*, pages 602–603, 1997.
11. C. Elliott. On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk Works. In A. F. R. Laboratory, editor, *Safe & Secure Systems and Software Symposium (S5)*, 2015.
12. C. Elliott. An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works. In A. F. R. Laboratory, editor, *Safe & Secure Systems and Software Symposium (S5)*, 2016.
13. P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah. StreamLAB: Stream-based monitoring of cyber-physical systems. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification*, pages 421–431, Cham, 2019. Springer International Publishing.
14. A. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, and J. A. Davis. SpeAR v2.0: Formalized past LTL specification and analysis of requirements. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 420–426, 2017.
15. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The jk ind model checker. In *International Conference on Computer Aided Verification*, pages 20–27. Springer, 2018.

16. S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. ARSENAL: automatic requirements specification extraction from natural language. In S. Rayadurgam and O. Tkachuk, editors, *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, volume 9690 of *Lecture Notes in Computer Science*, pages 41–46. Springer, 2016.
17. D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi. Formal requirements elicitation with FRET. In *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*, 2020.
18. D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann. Automated formalization of structured natural language requirements. *Inf. Softw. Technol.*, 137:106590, 2021.
19. A. Goodloe and L. Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.
20. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
21. K. Havelund and A. Goldberg. *Verify Your Runs*, pages 374–383. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
22. B. Jeannet and F. Gaucher. Debugging embedded systems requirements with STIMULUS: an automotive case-study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016.
23. J. C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550. ACM, 2002.
24. C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit. Provably correct reactive control from natural language. *Auton. Robots*, 38(1):89–105, jan 2015.
25. L. Lúcio, S. Rahman, C.-H. Cheng, and A. Mavin. Just formal enough? Automated analysis of EARS requirements. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 427–434, May 2017.
26. A. Mavridou, H. Bourbouh, P. L. Garoche, and M. Hejase. Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin cyber-physical challenge problems. Technical Report TM-2019-220374, National Aeronautics and Space Administration, February 2020.
27. A. Mavridou, H. Bourbouh, D. Giannakopoulou, T. Pressburger, M. Hejase, P.-L. Garoche, and J. Schumann. The ten Lockheed Martin cyber-physical challenges: Formalized, analyzed, and explained. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 300–310, 2020.
28. P. Moosbrugger, K. Y. Rozier, and J. Schumann. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51(1):31–61, 2017.
29. I. Perez, F. Dedden, and A. Goodloe. Copilot 3. Technical Report NASA/TM–2020–220587, NASA Langley Research Center, April 2020.
30. I. Perez, A. Goodloe, and W. Edmonson. Fault-tolerant swarms. In *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 47–54. IEEE, 2019.
31. I. Perez and A. E. Goodloe. Fault-tolerant functional reactive programming (extended version). *Journal of Functional Programming*, 30, 2020.

32. I. Perez, A. Mavridou, T. Pressburger, A. Goodloe, and D. Giannakopoulou. Artifact for Automated Translation of Natural Language Requirements to Runtime Monitors. `https://doi.org/10.5281/zenodo.5888956`. Accessed Jan 21, 2022.
33. I. Perez, A. Mavridou, T. Pressburger, A. Goodloe, and D. Giannakopoulou. Integrating FRET with Copilot: Automated Translation of Natural Language Requirements to Runtime Monitors. Technical Report NASA/TM–20220000049, NASA, January 2022.
34. I. Perez and H. Nilsson. Runtime verification and validation of functional reactive systems. *Journal of Functional Programming*, 30:e28, 2020.
35. L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS. Springer, November 2010.
36. L. Pike, N. Wegmann, S. Niller, and A. Goodloe. Copilot: Monitoring embedded systems. *Innov. Syst. Softw. Eng.*, 9(4):235–255, Dec. 2013.
37. T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–372, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
38. J. Schumann, P. Moosbrugger, and K. Y. Rozier. R2u2: monitoring and diagnosis of security threats for unmanned aerial systems. In *Runtime Verification*, pages 233–249. Springer, 2015.
39. H. Torfah. Stream-based monitors for real-time properties. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 91–110. Springer, 2019.
40. J. Wilmot. A core flight software system. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pages 13–14, New York, NY, USA, 2005. ACM.

MaskD: A Tool for Measuring Masking Fault-Tolerance*

Luciano Putruele[1,3](✉), Ramiro Demasi[2,3], Pablo F. Castro[1,3], and Pedro R. D'Argenio[2,3,4]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina, {lputruele,pcastro}@dc.exa.unrc.edu.ar

[2] Universidad Nacional de Córdoba, FAMAF, Córdoba, Argentina, {rdemasi,pedro.dargenio}@unc.edu.ar

[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

[4] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. We present MaskD, an automated tool designed to measure the level of fault-tolerance provided by software components. The tool focuses on measuring masking fault-tolerance, that is, the kind of fault-tolerance that allows systems to mask faults in such a way that they cannot be observed by the users. The tool takes as input a nominal model (which serves as a specification) and its fault-tolerant implementation, described by means of a guarded-command language, and automatically computes the masking distance between them. This value can be understood as the level of fault-tolerance provided by the implementation. The tool is based on a sound and complete framework we have introduced in previous work. We present the ideas behind the tool by means of a simple example and report experiments realized on more complex case studies.

1 Introduction

Fault-tolerance is an important characteristic of critical software. It can be defined as the capability of systems to deal with unexpected events, which may be caused by code bugs, interaction with an uncooperative environment, hardware malfunctions, etc. Examples of fault-tolerant systems can be found everywhere: communication protocols, hardware circuits, avionic systems, cryptocurrencies, etc. So, the increasing relevance of critical software in everyday life has led to a renewed interest in the automatic verification of fault-tolerant properties. However, one of the main difficulties when reasoning about these kinds of properties is given by their quantitative nature, which is true even for non-probabilistic systems. A simple example is given by the introduction of redundancy in critical systems. This is, by far, one of the most used techniques in fault-tolerance. In practice, it

* This work was supported by ANPCyT PICT-2017-3894 (RAFTSys), ANPCyT PICT 2019-03134, SeCyT-UNC 33620180100354CB (ARES), and EU Grant agreement ID: 101008233 (MISSION).

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 396–403, 2022.
https://doi.org/10.1007/978-3-030-99524-9_22

is well known that adding more redundancy to a system increases its reliability. Measuring this increment is a central issue for evaluating fault-tolerant software. On the other hand, there is no de-facto way to formally characterize fault-tolerant properties. Thus these properties are usually encoded using ad-hoc mechanisms as part of a general design.

The usual flow for the design and verification of fault-tolerant systems consists of defining a nominal model (i.e., the "fault-free" or "ideal" program) and afterwards extending it with faulty behaviors that deviate from the normal behavior prescribed by the nominal model. This extended model represents the way in which the system operates under the occurrence of faults. More specifically, a model extension enriches a transition system by adding new (faulty) states and transitions from and to those states, namely fault-tolerant implementation.

On the other hand, during the last decade, significant progress has been made towards defining suitable metrics or distances for diverse types of quantitative models including real-time systems [11], probabilistic models [7], and metrics for linear and branching systems [5,2,10,13,19]. Some authors have already pointed out that these metrics can be useful to reason about the robustness of a system, a notion related to fault-tolerance.

We present MaskD, an automated tool designed to measure the level of fault-tolerance among software components, described by means of a guarded-command language. The tool focuses on measuring masking fault-tolerant components, that is, programs that mask faults in such a way that they cannot be observed by the environment. It is often classified as the most beneficial kind of fault-tolerance and it is a highly desirable property for critical systems. The tool takes as input a nominal model and its fault-tolerant implementation and automatically computes the masking distance between them. It is based on a framework we have introduced in [4], and shown to be sound and complete. In Section 2 we give a brief introduction to this framework.

The tool is well suited to support engineers for the analysis and design of fault-tolerant systems. More precisely, it uses a computable masking distance function such that an engineer can measure the masking tolerance of a given fault-tolerant implementation, i.e., the number of faults that the implementation is able to mask in the worst case. Thereby, the engineers can measure and compare the masking fault-tolerance distance of alternative fault-tolerant implementations, and select one that best fits their preferences.

2 The MaskD Tool

MaskD takes as input a nominal model and its fault-tolerant implementation, and produces as output the masking distance between them, which is a value in the interval $[0, 1]$. The input models are described using the guarded command language introduced in [3], a simple programming language common for describing fault-tolerant algorithms. More precisely, a program is a collection of processes, where each process is composed of a collection of labelled actions of the style: $[\mathit{Label}]\ \mathit{Guard} \rightarrow \mathit{Command}$, where *Guard* is a Boolean condition over the actual

```
Process MEMORY {
 w : BOOL; // the last value written
 r : BOOL; // the value read from the
           // memory
 c0 : BOOL;
 Initial: w && c0 && r;
 [write1] true -> w=true, c0=true,
          r=true;
 [write0] true -> w=false, c0=false,
          r=false;
 [read0] !r -> r=r;
 [read1] r -> r=r;
}
```

```
Process MEMORY_FT {
 w : BOOL;
 r : BOOL;
 c0 : BOOL; // first bit
 c1 : BOOL; // second bit
 c2 : BOOL; // third bit
 Initial: w && c0 && c1 && c2 && r;
 [write1] true -> w=true, c0=true, c1=true,
          c3=true, r=true;
 [write0] true -> w=false, c0=false, c1=false,
          c3=false, r=false;
 [read0] !r -> r=r;
 [read1] r -> r=r;
 [fail1] faulty true -> c0=!c0, r =(!c0&&c1)||(c1&&c2)||
         (!c0&&c2);
 [fail2] faulty true -> c1=!c1, r =(c0&&!c1)||(!c1&&c2)||
         (c0&&c2);
 [fail3] faulty true -> c2=!c2, r =(c0&&c1)||(c1&&!c2)||
         (c0&&!c2);
}
```

Fig. 1. Processes for a memory cell example. On the left is the Nominal Model and on the right is the Fault-tolerant Model.

state of the program, *Command* is a collection of basic assignments, and *Label* is a name for the action. These syntactic constructions are called actions. The language also allows users to label an action as *internal* (i.e., silent actions). This is important for abstracting away internal parts of the system and building large models. Moreover, some actions can be labeled as *faulty* to indicate that they represent faults.

In order to compute the masking distance between two systems the tool uses notions coming from game theory. More precisely, a two-player game (played by the *Refuter* (R) and the *Verifier* (V)) is constructed using the two models. The intuition of this game is as follows. The Refuter chooses transitions of either the specification or the implementation to play, and the Verifier tries to match her choice. However, when the Refuter chooses a fault, the Verifier must match it with a masking transition. R wins if the game reaches the error state, denoted v_{err}. On the other hand, V wins when v_{err} is not reached during the game. Rewards are added to certain transitions in the game to reflect the fact that a fault was masked. Thus, given a play (a maximal path in the game graph) a function f_{mask} computes the value of the play: if it reaches the error state, the value is inversely proportional to the number of masking movements made by the Verifier; if the play is infinite, it receives a value of 0 indicating that the implementation was able to mask all the faults in the path. Summing up, the fault-tolerant implementation is masking fault-tolerant if the value of the game is 0. Furthermore, the bigger the number, the farther the masking distance between the fault-tolerant implementation and the specification.

As a running example, we consider a memory cell that stores a bit of information and supports reading and writing operations, presented in a state-based form in [6]. A state in this system maintains the current value of the memory cell ($m = i$, for $i = 0, 1$), writing allows one to change this value, and reading returns the stored value. In this system the result of a reading depends on the value stored in the cell. Thus, a property that one might associate with this model is

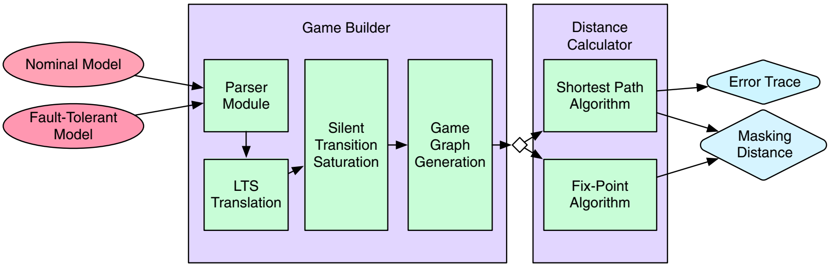


Fig. 2. Architecture of MaskD.

that the value read from the cell coincides with that of the last writing performed in the system.

A potential fault in this scenario occurs when a cell unexpectedly loses its charge, and its stored value turns into another one (e.g., it changes from 1 to 0 due to charge loss). A typical technique to deal with this situation is *redundancy*: in this case, three memory bits are used instead of only one. Writing operations are performed simultaneously on the three bits. Reading, on the other hand, returns the value that is repeated at least twice in the memory bits; this is known as *voting*. Figure 1 shows the processes representing the nominal and the fault-tolerant implementation of this example.

2.1 Architecture

MaskD is open source software written in Java. Documentation and installation instructions can be found at [1]. The architecture of the tool is shown in Fig. 2. We briefly describe below the key components of the tool:

Parser Module. It performs basic syntactic analysis over the input models, and produces data structures describing the inputs. Libraries Cup and JFlex were used to automatically generate the parser from the grammar describing the modeling language.

LTS Translation. The models obtained from the parser are translated into Labeled Transition Systems (LTSs), i.e., graphs whose vertices represent program states and whose transitions keep information about the actions in the models.

Silent Transition Saturation. The internal/silent transitions in the LTSs representing the input models are saturated using standard algorithms coming from process algebras [14]. As a result, saturated LTSs are generated, these are needed for verifying the masking relation when internal transitions are present.

Game Graph Generation. It uses the saturated LTSs to produce a game graph. Nodes in this graph encode the actual configuration of the game: the next player to play, the last played action, and references to the LTS states corresponding to the actual configuration of the game. Transitions in this graph correspond to the possible plays for the players, i.e., transitions in the original LTSs.

```
0. ERR_STATE
1. { <> , I_m1.read1 , <m1r,m1c0,m1c2> , V }
2. { <> , # , <m1r,m1c0,m1c2> , R }
3. { <> , I_m1.fail1 , <m1r,m1c0,m1c2> , V }
4. { <> , # , <m1c2> , R }
5. { <> , I_m1.fail3 , <m1c2> , V }
6. { <> , # , <> , R }
7. { <m1w,m1r,m1c0> , I_m1.write0 , <> , V }
8. { <m1w,m1r,m1c0> , # , <m1w,m1r,m1c0,m1c1,m1c2> , R }
```

Fig. 3. Error trace for the memory cell example.

Shortest Path Algorithm. If the input models are deterministic, Dial's shortest path algorithm is used to get the shortest path to the error state, from which the final value is calculated.

Fix-Point Algorithm. If the input models are non-deterministic, a bottom-up breadth-first search is used to compute the value of the game. This algorithm is based on well-known algorithms to solve reachability games that use *attractor* sets [15].

As explained above, an interesting point about our implementation is that, for deterministic systems, the masking distance between two systems can be computed by resorting to Dial's shortest path algorithm [17], which runs in linear time with respect to the size of the graphs used to represent the systems. In the case of non-deterministic systems, a fixpoint traversal approach based on breadth-first search is needed, making the algorithm less efficient. However, even in this case, the algorithm is polynomial.

2.2 Usage

The standard command to execute MaskD in a Unix operating system is:

```
./MaskD <options> <spec_path> <imp_path>
```

In this case the tool returns the masking distance between the specification and the implementation. Possible optional commands are: `-t: print error trace`, prints a trace to the error state; and `-s: start simulation`, starts a simulation from the initial state. A path to the error state is a useful feature for debugging program descriptions, which may be failing for unintended reasons. A trace for the memory cell example is shown in Fig. 3. States are denoted as `{spec_state, last_action_played, imp_state, player_turn}`. In this case, after two faults (bits being flipped), performing a read on the cell leads to the error state since on the nominal model the value is 0 while on the fault-tolerant model the value read by majority is 1. On the other hand, the simulation feature allows the user to manually select the available actions at each point of the masking game, which is also useful for verifying that the models behave as intended. By default, MaskD computes the masking distance for the given input using the algorithm for non-deterministic systems. The user can use option `-det` to switch to the deterministic masking distance algorithm.

Case Study	Redundancy	M. Distance	Time	Time(Det)
Redundant Memory Cell	3 bits	0.333	0.7s	0.6s
	5 bits	0.25	2.5s	1.9s
	7 bits	0.2	7.2s	5.7s
N-Modular Redundancy	3 modules	0.333	0.6s	0.5s
	5 modules	0.25	1.2s	0.7s
	7 modules	0.2	5.6s	3.8s
Dining Philosophers	2 philosophers	0.5	0.6s	0.6s
	3 philosophers	0.333	1.9s	0.9s
Byzantine Generals	3 generals	0.5	0.9s	–
	4 generals	0.333	17.1s	–
Raft LRCC (5)	1 follower	0	0.7s	0.8s
	2 followers	0	5.6s	3.6s
BRP (5)	1 retransm.	0.333	4.2s	–
	5 retransm.	0.143	4.8s	–
	10 retransm.	0.083	6.1s	–

Table 1. Some results of the masking distance for the case studies.

3 Experiments

We report on Table 1 some results of the masking distance for multiple instances of several case studies. These are: a Redundant Cell Memory (our running example), N-Modular Redundancy (a standard example of fault-tolerant system [18]), a variation of the Dining Philosophers problem [8], the Byzantine Generals problem introduced by Lamport et al. [12], the Log Replication consistency check of Raft [16], and the Bounded Retransmission Protocol (a well-known example of fault-tolerant protocol [9]) where we have modeled using silent actions and evaluating it with the weak masking distance. All case studies have been evaluated using the algorithms for both deterministic and non-deterministic games, with the exception of the non-deterministic models (i.e., the Byzantine Generals problem and the Bounded Retransmission Protocol). It is worth noting that most of the computational complexity arises from building the game graph rather than the actual masking distance calculation. For space reasons, we omit details of each case study and its complete experimental evaluation (delegated to the tool documentation).

Some words are useful to interpret the results of our running example. For the case of a 3 bit memory the masking distance is 0.333; the main reason for this is that the faulty model (in the worst case) is only able to mask 2 faults (in this example, a fault is an unexpected change of a bit) before failing to replicate the nominal behaviour (i.e., reading the majority value). Thus, the result comes from the definition of masking distance and taking into account the occurrence of two faults. The situation is similar for the other instances of this problem with more redundancy.

We have run our experiments on a MacBook Air with a 1.3 GHz Intel Core i5 processor and 4 GB of memory. The case studies for reproducing the results are available in the tool repository.

References

1. MaskD: Masking Distance Tool. https://doi.org/10.5281/zenodo.5815693
2. de Alfaro, L., Faella, M., Stoelinga, M.: Linear and branching system metrics. IEEE Trans. Software Eng. **35**(2), 258–273 (2009)
3. Arora, A., Gouda, M.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering **19**(11) (1993)
4. Castro, P.F., D'Argenio, P.R., Demasi, R., Putruele, L.: Measuring masking fault-tolerance. In: TACAS 2019, Prague, Czech Republic (2019)
5. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Simulation distances. Theor. Comput. Sci. **413**(1), 21–35 (2012)
6. Demasi, R., Castro, P.F., Maibaum, T.S.E., Aguirre, N.: Simulation relations for fault-tolerance. Formal Asp. Comput. **29**(6), 1013–1050 (2017)
7. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov processes. Theor. Comput. Sci. **318**(3), 323–354 (2004)
8. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica **1**(2), 115–138 (1971)
9. Groote, J.F., van de Pol, J.: A bounded retransmission protocol for large data packets. In: Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany, July 1-5, 1996, Proceedings. pp. 536–550 (1996)
10. Henzinger, T.A.: Quantitative reactive modeling and verification. Computer Science - R&D **28**(4), 331–344 (2013)
11. Henzinger, T.A., Majumdar, R., Prabhu, V.S.: Quantifying similarities between timed systems. In: Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings. pp. 226–241 (2005)
12. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
13. Larsen, K.G., Fahrenberg, U., Thrane, C.R.: Metrics for weighted transition systems: Axiomatization and complexity. Theor. Comput. Sci. **412**(28), 3358–3369 (2011)
14. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
15. nski, M.J.: Algorithms for solving parity games. In: Apt, K.R., Grädel, E. (eds.) Lectures in Game Theory for Computer Scientist, chap. 3, pp. 74–95. Cambridge University Press, New York, NY, USA (2011)
16. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference. pp. 305–319. USENIX Association (2014)
17. R.B.Dial: Algorithm 360: shortest-path forest with topological ordering. Communications of ACM **12** (1969)
18. Shooman, M.L.: Reliability of Computer Systems and Networks: Fault Tolerance,Analysis,and Design. John Wiley & Sons, Inc (2002)
19. Thrane, C.R., Fahrenberg, U., Larsen, K.G.: Quantitative analysis of weighted transition systems. J. Log. Algebr. Program. **79**(7), 689–703 (2010)

Better Counterexamples for Dafny

Aleksandar Chakarov[1], Aleksandr Fedchin[2](✉), Zvonimir Rakamarić[1], and Neha Rungta[1]

[1] Amazon Web Services, Seattle, WA, USA
aleksach,zvorak,rungta@amazon.com
[2] Tufts University, Medford, MA, USA
aleksandr.fedchin@tufts.edu

Abstract. Dafny is a verification-aware programming language used at Amazon Web Services to develop critical components of their access management, storage, and cryptography infrastructures. The Dafny toolchain provides a verifier that can prove an implementation of a method satisfies its specification. When the underlying SMT solver cannot establish a proof, it generates a counterexample. These counterexamples are hard to understand and their interpretation is often a bottleneck in the proof debugging process. In this paper, we introduce an open-source tool that transforms counterexamples generated by the SMT solver to a more user-friendly format that maps to the Dafny syntax and is suitable for further processing. This new tool allows the Dafny developers to quickly identify the root cause of a problem with their proof, thereby speeding up the development of Dafny projects.

Keywords: Dafny · Counterexample · Verification · SMT

1 Introduction

Dafny [12,11,6] is a verification-aware programming language popular in the automated reasoning community. Amazon Web Services (AWS), in particular, uses Dafny to develop critical components of their access management, storage, and cryptography infrastructures [5]. For these components, developers at AWS are writing Dafny programs that include the specification and the corresponding implementation. The advantage of using Dafny is that one can leverage the built-in verifier during the development process to automatically prove that the implementation of a method satisfies its specification. Finally, Dafny provides compilers for generating executable code in different target languages, such as C#, Java, and Go. For example, AWS developers have implemented the core AWS authorization logic in Dafny, and generated production Java code using a custom Java compiler. However, despite its advantages, Dafny has so far lacked in debugging functionality that could guide the developer to the root cause of a potential assertion (i.e., proof) failure. This was slowing down the developers, and it prompted the work on counterexample extraction that we present in this paper.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 404–411, 2022.
https://doi.org/10.1007/978-3-030-99524-9_23

To confirm that an assertion holds, Dafny verifier first translates Dafny source into the Boogie [1,3] intermediate verification language. Boogie generates a verification condition and submits it to an SMT solver (in our case Z3 [13,15]). When an assertion is violated, the solver provides a counterexample (i.e., a *counterexample model*). Understanding such counterexamples is key to debugging a failing proof. However, due to the two translation steps separating Dafny code from the SMT query, the counterexamples provided by the solver are difficult to understand and inhibit the debugging process. The scope of the problem becomes apparent from the fact that a counterexample extraction tool was once developed for Boogie [10], a language that is much closer to the solver in the verification pipeline than Dafny.

Prior attempts to present Dafny counterexamples in a human-readable format [9,8] have been successful with integers and Booleans but yielded unsatisfying results for other types. Our main contribution is a tool that improves the readability of Dafny counterexamples for other basic types, user-defined types, and collections. The tool converts a counterexample generated by the solver to a format that is intuitive to Dafny developers. In addition to improving the user experience, our tool lays the foundation for automatic test case generation, as we discuss in Section 4.

2 Motivation

Fig. 1 shows our running example of a Dafny program. The program defines a class `Simple` with an instance method `Match` that returns true if argument `s` (of type `string` that is alias for `seq<char>`) matches the pattern `p`. For simplicity, we only allow the `'?'` meta-character in the pattern, which matches any character. The program also includes specifications in the form of preconditions, postconditions, and loop invariants. The Dafny verifier uses these to prove the correctness of the method implementation.

To demonstrate the usefulness of counterexamples and the need to present them in a human-readable format, we introduce a bug into the `Match` method. We do this by deleting the part of the guard highlighted on line 16, thereby turning the method into a string equality check. The implementation of the method and its specification are no longer in agreement, and the Dafny verifier reports that the postcondition on line 7 might be violated on line 18. Even in this simple case, the information that the verifier gives, although it might help in localizing the problem, does not make the cause of the bug apparent. The counterexample provided by the solver spans hundreds of lines and is difficult to read. For example, Fig. 2 gives a slice of this counterexample showing just that variable `s` has type `seq<char>`.

In contrast, our tool, released with Dafny v3.3.0, generates the following counterexample that triggers the postcondition violation:

```
s:seq<char> = (Length := 1, [0] := 'A');
this:Simple = (p: @1);
@1:seq<char> = (Length := 1, [0] := '?');
```

```
class Simple
{
    var p:string

    method Match(s: string) returns (b: bool)
        requires |p| == |s|
        ensures b <==> forall n :: 0 <= n < |s| ==>
                s[n] == p[n] || p[n] == '?'
    {
        var i := 0;
        while i < |s|
            invariant i <= |s|
            invariant forall n :: 0 <= n < i ==>
                s[n] == p[n] || p[n] == '?'
        {
            if s[i] != p[i] && p[i] != '?'
            {
                return false;
            }
            i := i + 1;
        }

        return true;
    }
}
```

Fig. 1: A Dafny program that matches a string against a pattern. The highlighted code is removed to introduce a bug as described in Section 2.

Here, the first line indicates that argument `s` is a sequence of characters (i.e., a string) of length 1, where the character at index 0 is `A`. Field `p` of the receiving object (`this`) points to object `@1`, where `@1` is a string of length 1 with the `?` meta-character at index 0. With these inputs, the buggy implementation of method `Match` returns false because the pattern and argument are not identical, even though they should match according to the specification.

Before we incorporated our tool into Dafny, it would report the following counterexample for this same program:

```
s = [Length 1](T@U!val!71); this = (T@U!val!75);
```

Clearly the counterexample generated by our tool is much more informative. Among the tools in this space that we know of, only Why3 [7] has counterexample generation functionality of similar complexity.

```
s#0 -> T@U!val!71 // Boogie variable s#0 has ID 71
BoxType -> T@T!val!15 // Boogie's Box type has ID 15
type -> { // The Boogie type of variable s#0 has ID 22
  T@U!val!71 -> T@T!val!22
}
SeqTypeInv0 -> { // Boogie type of s#0 is Seq Box:
  T@T!val!22 -> T@T!val!15
}
$Is -> { // Dafny type of variable s#0 has ID 76
  T@U!val!71 T@U!val!76 -> true
}
Tag -> { // Type with ID 76 is a subtype of a type with ID 13
  T@U!val!76 -> T@U!val!13
}
TagSeq -> T@U!val!13 // Dafny type with ID 13 is seq
TChar -> T@U!val!1 // Dafny type with ID 1 is char
Inv0_TSeq -> { // Dafny type with ID 76 is seq<char>
  T@U!val!76 -> T@U!val!1
}
```

Fig. 2: An extract of a counterexample model generated by Z3 for the code in Fig. 1 that shows that variable `s` has type `seq<char>`.

3 Design and Implementation

We implemented our tool on top of the existing Dafny counterexample extraction functionality by adding key new features such as the ability to extract types from the Z3 model and support complex types (e.g., sequences) beyond just integers and Booleans. Our type extraction supports type parameterization and type renaming, and makes extracted counterexamples useful beyond improved user experience, e.g., automatic test case generation (see Section 4).

We illustrate how the counterexample generation tool works using our running example from Fig. 1. Before the tool can look up the types and values of specific variables, it must first identify the variables and program states[1] relevant to the given counterexample. In our example, there are four relevant program states: the initial state, the state following the initialization of `i`, the state at the loop head, and the state preceding the return statement. There are three relevant variables: `this`, `s`, and `i`. Our tool inherits the extraction of this information from the Z3 model from the existing counterexample generator.

Once we identify the relevant variables and states, we determine the type of each variable. This is a two-step process. First, we extract the Boogie type of a variable in the Boogie translation from the Z3 model (e.g., `Seq Box` for `s` in Fig. 2). Then, we map it to its corresponding Dafny type (`seq<char>` for `s` in

[1] Dafny to Boogie translator marks Dafny program states with the `:capturedState` annotation in Boogie.

Variable	Constraint	Counterexample
`b:bv6`	`b == 1`	`b:bv6 := 0`
`r:real`	`r != 0.2`	`r:real := 1.0/5.0`
`c:char`	`c != 'c'`	`c:char := 'c'`
`c:char`	`c == 'c'`	`c:char := 'A'`
`d:M.DType`	`d.i > 4`	`d:M.DType = A(i := -34)`
`a:array2?<int>`	`a.Length0 < 2 || a.Length1 < 2 || a[1,1] != 3`	`a:_System.array2?<int> := (Length0 := 2, Length1 := 40, [1,1] := 3)`
`s:set<int>`	`1 in s`	`s:set<int> = {1 := false}`
`s:set<int>`	`1 !in s`	`s:set<int> = {1 := true}`
`s:seq<int>`	`|s| < 1 || s[0] != 3`	`s:seq<int> = [3]`
`s:seq<int>`	`|s| < 2 || s[1] != 3`	`s:seq<int> = (Length := 2, [1] := 3)`
`m:map<int, char>`	`1 !in m`	`m:map<int,char> = (1 := 'A', 2 :='B', 3 :='C', 4 :='D')`

Table 1: Counterexamples generated for different constraints.

Fig. 2). The latter step may require choosing among the different types listed by the model (e.g., between `string` and `seq<char>`). We give preference to the original type names (`seq<char>`) to clearly separate user-defined from built-in types. We also take special care to extract type parameters and reconstruct the Dafny type name from its Boogie translation, for example, `Module.Module2.Class` from `Module_mModule2.Class`.

After determining the type of a variable, our tool extracts the string representation of the variable's value. The way the value is specified in the counterexample model depends on the variable type. In method `Match` in Fig. 1, the receiver is an instance of a user-defined class `Simple`, so the tool looks up the value of its only field `this.p`. This field is itself a non-primitive variable, and so we recurse into its definition until we reach a value of a primitive type, which we then use to construct the non-primitive value. In case the model does not specify a value for some variable of primitive type, the tool automatically generates an adequate value that is different from any other value of that type in the model or source code.

Our implementation of the counterexample extraction tool supports all basic types, user-defined classes, datatypes, arrays, and the three most commonly used collections (sequence, sets, and maps). See Table 1 for concrete examples of the tool's output. Previously, the counterexample generator could only show the values of integer and Boolean variables, constructor names used to create a datatype, or the length of a sequence. The differences between our new implementation and past versions are mostly due to the support we added for new types and collections (e.g., chars, bit vectors, maps). However, we also had to revamp and bring up-to-date some of the previously implemented features that have since ceased to function as intended. For instance, Krucker and Schaden [9] show that they could once extract the values of object's fields, but this functionality had not been maintained and it stopped working properly. We speculate that the lack of automated testing likely contributed to the failure to adapt the counterexample extraction to the rapidly evolving Dafny infrastructure. To ensure maintainability, we have developed an extensive test suite as part of this work. The test suite contains 54 tests covering all supported types and collections, and is executed as part of the continuous integration process of Dafny.

To benefit from the counterexample extraction feature while working in Visual Studio Code IDE, the user needs only to install the Dafny plugin.[2] In addition to visualizing counterexamples in the VS Code plugin, the counterexample extraction tool provides a public API and can be imported as a dependency by any C# project. Finally, we made our accompanying artifact publicly available to improve the reproducibility of our contributions [4].

4 Conclusions and Future Work

This paper presents the new, improved version of Dafny's counterexample extraction tool, which now extracts values of all variables of basic or user-defined types as well as variables representing algebraic datatypes, arrays, sequences, sets, and maps. We integrated the tool into the Dafny plugin for Visual Studio Code, and released it with Dafny v3.3.0. The tool has already been used by Dafny developers to assist them during the proof debugging process.

Note that a counterexample reported by the Dafny verifier might occasionally be a spurious one. This is a well-known problem that users of these verifiers struggle with. It is typically due to the incompleteness of the underlying SMT solver, for example, in the presence of quantifiers. A possible solution to identifying spurious counterexamples is to generate a concrete test case from the counterexample, execute the program concretely using the test case, and observe whether the concrete execution violates the same property [2,14]. The counterexample extraction tool presented in this paper, with its ability to extract the type and concrete value of any variable, can be used for test case generation as well. As future work, we plan to build on this functionality and implement extensions for identifying spurious counterexamples as well as for automatic unit test generation.

Acknowledgements We thank Christoph Amrein, Rustan Leino, Bryan Parno, Shaz Quadeer, Robin Salkeld, and Remy Wilems for reviewing our pull requests to Boogie and Dafny, as well as for their feedback that helped us to improve the early versions of the tool. We also thank Sean McLaughlin, Matthias Schlaipfer, and Jenny Xiang for their insights into the usability of the tool in practice, and to Jeff Foster for reviewing the initial drafts of the paper.

[2] Developed by the Correctness Lab at OST Eastern Switzerland University of Applied Sciences [9,8].

References

1. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387 (2005). https://doi.org/10.1007/11804192_17
2. Becker, B.F.H., Lourenço, C.B., Marché, C.: Explaining counterexamples with giant-step assertion checking. In: Workshop on Formal Integrated Development Environment. EPTCS, vol. 338, pp. 82–88 (2021). https://doi.org/10.4204/EPTCS.338.10
3. Boogie, https://github.com/boogie-org/boogie
4. Chakarov, A., Fedchin, A., Rakamarić, Z., Rungta, N.: Better counterexamples for Dafny artifact (2021). https://doi.org/10.5281/zenodo.5571033
5. Cook, B.: Formal reasoning about the security of Amazon web services. In: International Conference on Computer Aided Verification. pp. 38–47 (2018). https://doi.org/10.1007/978-3-319-96145-3_3
6. Dafny, https://github.com/dafny-lang/dafny
7. Dailler, S., Hauzar, D., Marché, C., Moy, Y.: Instrumenting a weakest precondition calculus for counterexample generation. Journal of Logical and Algebraic Methods in Programming **99**, 97–113 (2018). https://doi.org/10.1016/j.jlamp.2018.05.003
8. Hess, M., Kistler, T.: Dafny Language Server Redesign. Term project, HSR Hochschule für Technik Rapperswil (2019)
9. Krucker, R., Schaden, M.: Visual Studio Code Integration for the Dafny Language and Program Verifier. Bachelor's thesis, HSR Hochschule für Technik Rapperswil (2017)
10. Le Goues, C., Leino, K.R.M., Moskal, M.: The Boogie verification debugger (tool paper). In: International Conference on Software Engineering and Formal Methods. pp. 407–414 (2011). https://doi.org/10.1007/978-3-642-24690-6_28
11. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370 (2010). https://doi.org/10.1007/978-3-642-17511-4_20
12. Leino, K.R.M.: Accessible software verification with Dafny. IEEE Software **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Nilizadeh, A., Calvo, M., Leavens, G.T., Le, X.B.D.: More reliable test suites for dynamic APR by using counterexamples. In: IEEE International Symposium on Software Reliability Engineering (2021), to appear
15. Z3, https://github.com/Z3Prover/z3

Constraint Solving

cvc5: A Versatile and Industrial-Strength SMT Solver*

Haniel Barbosa[3], Clark Barrett[1], Martin Brain[4], Gereon Kremer[1], Hanna Lachnitt[1], Makai Mann[1], Abdalrhman Mohamed[2], Mudathir Mohamed[2], Aina Niemetz[1](✉), Andres Nötzli[1], Alex Ozdemir[1], Mathias Preiner[1], Andrew Reynolds[2], Ying Sheng[1], Cesare Tinelli[2], and Yoni Zohar[1,5]

[1] Stanford University, Stanford, USA (✉)`niemetz@cs.stanford.edu`
[2] The University of Iowa, Iowa City, USA
[3] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[4] City, University of London, London, UK
[5] Bar-Ilan University, Ramat Gan, Israel

Abstract. cvc5 is the latest SMT solver in the cooperating validity checker series and builds on the successful code base of CVC4. This paper serves as a comprehensive system description of cvc5's architectural design and highlights the major features and components introduced since CVC4 1.8. We evaluate cvc5's performance on all benchmarks in SMT-LIB and provide a comparison against CVC4 and Z3.

Keywords: automated reasoning · constraint solving · satisfiability modulo theories · cvc5

1 Introduction

SMT solvers are widely recognized as crucial back-end reasoning engines for a variety of applications, including software and hardware verification [19, 52, 60, 68, 82, 86], model checking [41, 42, 98], type checking, static analysis, security [10, 62], automated test-case generation [40, 135], synthesis [2, 65], planning, scheduling, and optimization [127]. Notable SMT solvers include Bitwuzla [92], Boolector [98], CVC4 [21], MathSAT [46], OpenSMT2 [72], SMTInterpol [44], SMT-RAT [50], STP [61], veriT [35], Yices2 [55], and Z3 [90].

Among these, the family of *cooperating validity checker (CVC)* tools [21, 26, 27, 132] have played an important role, both in research and in practice [11, 48, 70, 137, 138]. The most recent incarnation, CVC4, was a from-scratch rewrite of

* This work was supported by AFOSR, AFRL, Amazon Web Services, BSF, Certora, DARPA, ERC, GE Global Research, Google, Intel, Meta, NASA, NSF, ONR, SRC, United Technologies Research Center, and Stanford University—including the Center for Automated Reasoning (Centaur), the Center for Blockchain Research, the Agile Hardware Center (AHA), and the SystemX Alliance. More details can be found at: https://cvc5.github.io/acknowledgements.html.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 415–442, 2022.
https://doi.org/10.1007/978-3-030-99524-9_24

CVC3, written with the aim of creating a flexible and performant architecture that could last far into the future. The fact that CVC4 has integrated over a decade's worth of SMT research and development while becoming increasingly robust and performance-competitive attests to the success of that endeavor.

In this paper, we introduce cvc5, the next solver in the series. cvc5 is not a rewrite of CVC4 and indeed builds on its successful code base and architecture. Compared to other SMT solvers, cvc5 supports a diverse set of theories (all standard SMT-LIB theories, and many non-standard theories) and features beyond regular SMT solving such as higher-order reasoning and syntax-guided synthesis (SyGuS) [3]. The name-change[6] rather acknowledges both a (mostly) new team of developers as well as the significant evolution the tool has undergone since CVC4 was described in a tool paper published in 2011 [21]. Moreover, cvc5 comes with updated documentation, new and improved APIs, and more user-friendly installation. Most importantly, it introduces several significant new features. Like its predecessors, cvc5 is available under the 3-clause BSD open source license and runs on all major platforms (Linux, macOS, and Windows).

We make the following contributions:

- An in-depth description of the architectural design of cvc5 and how its pieces and modules work together.
- A comprehensive summary of all features that have been added to the solver since CVC4 was introduced in [21].
- A description of major features introduced since CVC4 1.8, the final version of CVC4, including:
  - a new C++ API, and new Python and Java APIs that build on top of it;
  - a new theory solver for the theory of fixed-size bit-vectors;
  - a new and extensive proof-production module;
  - a new procedure for non-linear arithmetic; and
  - a syntax-guided quantifier-instantiation procedure [96].
- Evidence, based on experimental evaluation and industrial use cases, that cvc5 is in fact both *versatile* and *industrial-strength*.

2 Architecture and Core Components

cvc5 supports reasoning about quantifier-free and quantified formulas in a wide range of background theories and their combinations, including all theories standardized in SMT-LIB [22]. It further natively supports several non-standard theories and theory extensions. These include, among others, separation logic, the theory of sequences, the theory of finite sets and relations, and the extension of the theory of reals with transcendental functions.

In this section, we start with a brief overview of the core components of cvc5, and then discuss them in more detail in the following subsections. A high-level overview of the system architecture is given in Figure 1.

[6] Whereas the convention for previous solvers in the CVC family was to use capital letters, here we introduce a new convention of using lower-case letters (or alternatively small capitals, as in this paper, which we find to be more visually appealing).

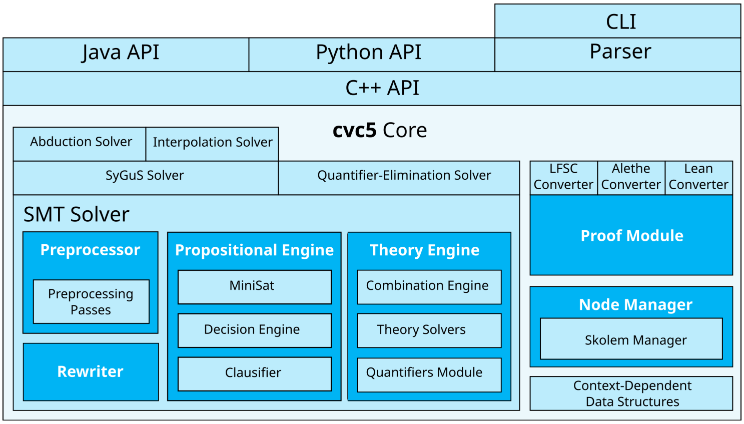


Fig. 1: High-level overview of cvc5's system architecture.

The central engine of cvc5 is the *SMT Solver* module, which is based on the CDCL($\mathcal{T}$) framework [99] and relies on a customized version of the MiniSat propositional solver [57] at its core. The SMT Solver consists of several components: the Rewriter and the Preprocessor modules, which apply simplifications locally (at the term level) and globally (on the whole input formula), respectively; the Propositional Engine, which serves as a manager for the CDCL($\mathcal{T}$) SAT solver; and the Theory Engine, which manages theory combination and all theory-specific and quantified reasoning procedures.

Besides standard satisfiability checking, cvc5 provides additional functionality such as abduction, interpolation, syntax-guided synthesis (SyGuS) [3], and quantifier elimination. Each of these features is implemented as an additional solver built on top of the SMT Solver. The SyGuS Solver is the main entry point for synthesis queries, which encode SyGuS problems as (higher-order) satisfiability problems with both semantic and syntactic constraints [114]. The Quantifier Elimination Solver performs quantifier elimination based on tracking the quantifier instantiations of the SMT Solver [116]. The Abduction Solver and the Interpolation Solver are both SyGuS-based [110] and thus are built as layers on top of the SyGuS Solver.

cvc5 provides a C++ API as the main interface, not just for external client software, but also for its own parser and for additional language bindings in Java and Python. cvc5 also provides a textual command-line interface (CLI), built on top of the parser, which supports SMT-LIBv2 [25], SyGuS2 [104] and TPTP [134] as input languages. The Proof Module can output formal unsatisfiability proofs in three proof formats: Alethe [128], Lean 4 [88], and LFSC [133].

2.1 The SMT Solver Module

The SMT Solver module is the centerpiece of cvc5 and is responsible for handling all SMT queries. Its functionality includes, in addition to satisfiability checking, constructing models for satisfiable input formulas and extracting assumptions, cores, and proof objects for unsatisfiable formulas. The main components of the SMT Solver module are described below.

Preprocessor. Before any satisfiability check, cvc5 applies to each formula from an input problem a sequence of satisfiability-preserving transformations. We distinguish between (i) required *normalization* passes, e.g., removal of ite terms; (ii) optional *simplification* passes aimed at making the formula easier to solve, e.g., finding entailed theory literals; and (iii) optional *reduction* passes that transform the formula from one logic to another, e.g., from non-linear integer arithmetic to a bit-vector problem with configurable bit-width. Currently, cvc5 implements 34 passes, executed in a fixed order. Optional passes can be enabled and disabled via configuration options. Preprocessing passes are self-contained, and adding or modifying passes does not require knowledge of the internals of the SMT solver engine.

Propositional Engine. The Propositional Engine serves as the core CDCL($\mathcal{T}$) engine [99], which takes the Boolean abstraction of the input formula (together with any lemmas produced during solving) and produces a satisfying assignment for that abstraction. Its main components are the Clausifier and the propositional satisfiability (SAT) solver. The Clausifier converts the Boolean abstraction into Conjunctive Normal Form (CNF), which then serves as input for the SAT solver. In cvc5, as in CVC4, we use a customized version of MiniSat [57] as the core SAT solver. Extensions we have added to MiniSat include: the production of resolution proofs; native support for pushing and popping assertions; and a *Decision Engine* [12], which can be used to create customized decision heuristics for MiniSat.

During its search, the Propositional Engine asserts a theory literal $(\neg)p$ to the Theory Engine as soon as the SAT solver assigns a truth value to the propositional variable abstracting the atom p. We refer to the set of all such literals as the *currently asserted literals*. When checking the consistency of the set L of currently asserted literals in the overall background theory $\mathcal{T}$, we distinguish between two levels of effort: *standard* and *full*, depending on whether the SAT solver has a partial or full model, respectively, for the Boolean abstraction. At standard effort, a theory solver may optionally perform some lightweight consistency checking. At full effort, the theory solver must either produce a lemma (following the splitting-on-demand approach [23]) or determine whether L is satisfiable or not and, in the latter case, produce a *conflict clause*, a clause that is valid in the theory $\mathcal{T}$ but is inconsistent with L.

Rewriter. The Rewriter module is responsible for converting terms via a set of rewrite rules into semantically equivalent normal forms. In contrast to preprocessing, rewriting is done *during solving*. In fact, all major components of cvc5 invoke the Rewriter to ensure that the terms they work with are normalized, thereby simplifying their implementation. Rewrite rules are applied locally, i.e., independent of the currently asserted literals, and are divided into required and optional rules, of which the latter can be enabled or disabled by the user. The Rewriter maintains a cache to avoid processing any term more than once.

Examples of rewrites include simplifications such as $x + 0 \rightsquigarrow x$, normalizations that sort the operands of associative and commutative operators, and operator eliminations such as $x \leq y \rightsquigarrow y + 1 > x$ (when x and y have integer sort). In certain contexts, e.g., enumerative SyGuS approaches, aggressive rewriting rules, which would be detrimental to SMT solving, can be beneficial. Such rules are implemented in an *Extended Rewriter*, which is enabled when needed.

To help automate improvements to the Rewriter, we developed a workflow that detects and enumerates new rewrite rule candidates using the SyGuS solver [101]. It works by detecting and suggesting *critical pairs*, i.e., pairs of equivalent terms that are not rewritten to the same term by the current rules.

Theory Engine. The Theory Engine is the main entry point for checking the theory consistency of the theory literals asserted by the Propositional Engine. It dispatches each of these literals to the appropriate theory solvers and is further responsible for dispatching any propagated literals or lemmas generated by the theory solvers back to the Propositional Engine.

When multiple theory solvers are enabled, the Combination Engine submodule is responsible for coordinating between them. Like CVC4, cvc5 uses the *polite* theory combination mechanism [74,108,130]. This includes propagating or performing case splits on equalities and disequalities between *shared* terms (terms appearing in the literals of more than one theory solver). As in CVC4, the algorithm for computing these splits is based on care graphs [75].

The Combination Engine controls the Model Manager, which is responsible for combining models from multiple theories and constructs a model for the input formula. The Model Manager also maintains an equivalence relation E over all the terms in the input formula, induced by all of the currently asserted literals that are equalities. When invoked, the Model Manager has the responsibility of assigning concrete values to each equivalence class of E with the assistance of the individual theory solvers, which provide values for terms in their theory. Typically, the Model Manager is invoked only when the theory solvers have reached a saturation point that allows the Theory Engine to conclude that the input problem is satisfiable (and thus, a model can be constructed successfully).

As in CVC4, each sub-formula of the input that starts with a quantifier is abstracted by a propositional variable. When any such variable or its negation is asserted, the Theory Engine dispatches the corresponding quantified formula to the Quantifiers Module, which generates suitable quantifier instantiations. Since certain techniques for handling quantified formulas, e.g., E-matching [89], require knowledge of the state and terms known by the other theory solvers, this module has access to *all* equality information from all theory solvers.

Theory Solvers. cvc5 supports a wide range of theories, including all theories standardized in SMT-LIB. Each theory solver relies on an Equality Engine Module, which implements congruence closure over a configurable set of operators, typically those that belong to the solver's theory. The Equality Engine is responsible for quickly detecting conflicts due to equality reasoning. In addition, all theories communicate reasoning steps to the rest of the system via the Theory Inference Manager. Every theory solver emits lemmas, conflict clauses, and propagated literals through this interface. The Theory Inference Manager implements or simplifies common usage pattern like caching and rewriting lemmas, proof construction, and collection of statistics. Every lemma or conflict sent from a theory is associated with a unique identifier for its kind, the *inference identifier*, which is a crucial debugging aid. Below, we briefly survey the theory solvers in cvc5, along with their main implementation techniques.

Linear Arithmetic. The linear arithmetic solver [78] extends the simplex procedure adapted for SMT by Dutertre and de Moura [56]. It implements a sum-of-infeasibilities-based heuristic [79], an integration with the external GLPK LP solver [80], and certain heuristics proposed by Griggio [63]. Integer problems are handled by solving their real relaxation before using branching [64] and cutting planes [54] to find integer solutions. The branch-and-bound method optionally generates lemmas consisting of ternary clauses inspired by *unit-cube* tests [39].

Non-linear Arithmetic. For non-linear arithmetic problems, cvc5 resorts to linear abstraction and refinement. It uses a combination of independent sub-solvers integrated with the linear arithmetic solver and invoked only when the linear abstraction is satisfiable. One sub-solver implements cylindrical algebraic coverings [1], while the other sub-solvers are based on incremental linearization [45]. A variety of lemma schemas are used to assert properties of non-linear functions (e.g., multiplication and trigonometric functions) in a counterexample-guided fashion [123]. Non-linear integer problems are solved by incremental linearization and incomplete techniques based on reductions to bit-vectors.

Arrays. As in CVC4, the array solver is based on a decision procedure by de Moura and Bjørner [91] but following the more detailed description by Jovanović and Barrett [75]. An alternative experimental implementation based on an approach by Christ and Hoenicke [43] is also available.

Bit-Vectors. For the theory of fixed-size bit-vectors, cvc5's main approach is *bit-blasting*, which refers to the process of translating bit-vector problems into equisatisfiable SAT problems, and is applied after preprocessing. In cvc5, we distinguish two modes for bit-blasting: *lazy* and *eager*. Lazy bit-blasting seamlessly integrates with the CDCL($\mathcal{T}$) infrastructure of cvc5 and fully supports the combination of bit-vectors with any theory supported by cvc5. It further leverages the full power of cvc5's Equality Engine for reasoning about equalities over bit-vector terms and also uses the solve-under-assumptions feature [57] supported by many state-of-the-art SAT solvers. For problems that can be fully reduced to bit-vectors, cvc5 can also be used in eager mode. This mode does not rely on solving under assumptions, but instead directly asserts all of the bit-blasted constraints to the SAT solver, which usually enables more simplifications. Additionally, cvc5 supports the Ackermannization and eager bit-blasting of constraints involving uninterpreted functions and sorts [66].

Datatypes. For quantifier-free constraints over datatypes, we use a rule-based procedure that follows calculi already implemented in CVC4 [24, 112] and that optimizes the sharing of selectors over multiple constructors [125].

Floating-Point Arithmetic. Formulas in the theory of floating-point arithmetic are translated to equisatisfiable formulas in the theory of bit-vectors, in a process referred to as *word-blasting*. For this, cvc5 integrates the SymFPU [37] library, which was first used in CVC4 and has also been integrated in the Bitwuzla SMT solver [92]. This approach admits several optimizations compared to earlier solvers, which translate directly to the *bit-level*, e.g., CNF or AIGs. Another difference from older approaches [38] is that translation is done at the formula level instead of the term level. Conversions between real and floating-point terms are treated as uninterpreted functions and refined if the models of the real arithmetic and the floating-point solver do not agree. The refinement lemmas use the monotonicity of the conversion functions to constrain the floating-point and real arithmetic terms to matching intervals that exclude the current model.

Sets and Relations. cvc5 implements a solver for the parametric theory of finite sets, i.e., sets whose elements are of any sort supported by cvc5. The core decision procedure for sets is extended with support for cardinality constraints [13]. The set theory solver is extended with a sub-module that specializes in relational constraints [87], where relations are modeled as sets of tuples.

Separation Logic. In separation logic, the semantics of constraints assume a location and data type for specifying the model of the heap. cvc5 supports an extension of the SMT-LIB language for separation logic [73], in which the location and data types of the heap can be any sort supported by cvc5. The classical separation logic connectives are treated as theory predicates which are lazily reduced to constraints over sets and uninterpreted functions [115].

Strings and Sequences. For strings and sequences, cvc5 implements a solver consisting of multiple layered components. At its core, the solver reasons about length constraints and word equations [84], supplemented with reasoning about code points to handle conversions between strings and integers efficiently [119]. Extended functions such as string replacement are lazily reduced to word equations after context-dependent simplifications [126]. When necessary, the regular expressions in input problems are unfolded and derivatives are computed [85]. The string theory solver further incorporates aggressive simplification rules that rely on abstractions to derive facts about string terms [118]. Finally, conflicts are detected eagerly on partial assignments from the SAT solver by computing the congruence closure and constant prefixes and suffixes of string terms.

Uninterpreted Functions. The theory of uninterpreted functions is handled in largely the same way as in CVC4. It follows Simplify's approach [53] extended with support for fixed finite cardinality constraints [121]. This extension is used in combination with finite-model-finding techniques for finding finite models based on minimal interpretations of uninterpreted sorts.

Quantifiers. Quantified formulas are all handled by the Quantifiers Module, which resembles a theory solver. The module contains many sub-solvers, all based on some form of quantifier instantiation, and each specializing in solving specific classes of quantified formulas. The Quantifiers Module relies on heuristic E-matching when uninterpreted functions are present [89]. This technique is supplemented by conflict-based instantiation for detecting when an instantiation is in conflict with the currently asserted literals [16,124]. The Quantifiers Module additionally incorporates finite-model-finding techniques, which are useful for detecting satisfiable input problems [122]. It also relies on enumerative approaches when other techniques are incomplete [109]. For quantifiers over linear arithmetic, it uses a specialized counterexample-guided based approach for quantifier instantiation [116]. An extension of this technique is used for quantified bit-vector logics [95]. For other quantified logics in pure background theories, e.g., over floating-point or non-linear arithmetic, cvc5 relies on syntax-guided quantifier instantiation [96]. The Quantifiers Module also contains sub-solvers implementing more advanced solving paradigms, including: a module for doing Skolemization with inductive strengthening and enumeration of sub-goals for inductive theorem proving problems [117], a finite-model-finding technique for recursive functions [113], and a solver for syntax-guided synthesis [114].

2.2 Proof Module

The Proof Module of cvc5 was built from scratch and replaces the proof system of CVC4 [67,77], which was incomplete and suffered from a number of architectural shortcomings. The design of cvc5's proof module was guided by the following principles. First, the overhead incurred by proof production should be at most linear in the solving time. Second, the emitted proofs should be detailed enough to enable efficient (i.e., polynomial) checking, ensuring that proof checking is inherently simpler than solving. Third, disabling a system component when in proof production mode because it lacks adequate proof generation capabilities should be done rarely and only if the component is not crucial for performance. Finally, given the different needs of users and the trade-offs offered by different proof systems, proof production should be flexible enough to allow the emission of proofs in different formats.

Following these design principles, the Proof Module in cvc5 produces detailed proofs for nearly all of its theories, rewrite rules, preprocessing passes, internal SAT solvers, and theory combination engines. It further supports eager and lazy proof production with built-in proof reconstruction. This enables proof production for some notoriously challenging functionalities, such as substitution and rewriting (common, for example, in simplification under global assumptions and in string solving [126]). Furthermore, although it maintains internally a single proof representation, cvc5 is able to emit proofs in multiple formats, including those supported by the LFSC [133] proof checker and the Lean 4 [88], Isabelle/HOL [100] and Coq [30] proof assistants.

2.3 Node Manager

Formulas and terms are represented uniformly in cvc5 as nodes in a directed acyclic graph, reference-counted and managed by the Node Manager. The Node Manager further maintains a Skolem Manager, which is responsible for tracking

Skolem symbols introduced during solving. All cvc5 instances in the same thread share the same Node Manager instance.

Nodes are immutable and are aggressively shared using *hash consing*: whenever a new node is about to be created, the Node Manager checks whether a node with the same structure already exists, and if it does, it returns a reference to the existing node instead. Besides saving memory, this ensures that syntactic equality checks can be performed in constant time (by comparing the unique ids assigned to each node). Reference counting allows the Node Manager to determine when to dispose of nodes. Weak references are used whenever possible to limit the overhead of reference counting.

Nodes store 96 bits of metadata (id, reference count, kind, and number of children) and a variable number of pointers to child nodes. The kind of a node can be an operator kind, e.g., addition, or a leaf kind, e.g., a variable. Optional additional static information associated with nodes can be stored separately in hash maps referred to as *node attributes*. Since node attributes are managed by the Node Manager, which may be shared by multiple solver instances, attributes must only be used to capture inherent node properties (i.e., properties that are independent of run-time options).

Many theory solvers, including those for quantifiers, strings, arrays, nonlinear arithmetic, and sets, introduce terms with Skolem (i.e., fresh) constants during solving. Such constants are centrally generated by the Skolem Manager, which also associates with each of them a term of the same sort, the constant's *witness form*. If the computed witness form for a constant matches that of a previously used constant, the previous constant can be reused. This not only provides a deterministic way of generating fresh constants during solving but also allows the system to minimize the number of introduced constants. This reuse is crucial for performance in some theory solvers [120].

2.4 Context-Dependent Data Structures

Certain applications of SMT solvers require multiple satisfiability checks with similar assertions. To support such applications, the SMT-LIB standard includes commands to save (with a *push* command) the current set of user-level assertions and restore (with a *pop* command) a previous set. This allows the solver to reuse parts of the work from earlier satisfiability checks and amortizes startup cost. Most of the state of cvc5 depends directly or indirectly on the current set of assertions. So whenever the user pushes or pops, cvc5 has to save or restore the corresponding state. Similarly, whenever the SAT solver makes a decision or backtracks to a previous decision point, each theory solver has to save or restore the corresponding information.

To support these operations, cvc5 defines a notion of *context level*, which increases with each push and decreases with each pop operation, and implements *context-dependent data structures*. These data structures behave similarly to corresponding mutable data structures provided in the C++ standard library, except that they are associated with a context level and automatically save and restore their state as the context increases or decreases. For efficiency reasons,

```
s = Solver()
i = s.getIntegerSort()
x = s.mkConst(i, "x")
s.assertFormula(
  s.mkTerm(kinds.Equal,
    s.mkTerm(kinds.Mult,
      x, s.mkInteger(2)),
    s.mkInteger(4)))
s.checkSat()
```

(a) The base cvc5 Python API

```
solve(2 * Int("x") == 4)
```

(b) The “pythonic” API

Fig. 2: Example of using the Python APIs of cvc5.

this state data is stored using a region-based custom allocator that allocates one region per context level, allowing all state data associated with a level to be freed simultaneously by simply freeing the corresponding region.

3 Highlighted Features

In this section, we discuss features that are new in cvc5 as well as some of the more prominent user- and developer-facing features. We compare them to their counterparts in CVC4 when applicable.

Application Programming Interfaces (APIs). cvc5 provides a lean, comprehensive, and feature-complete C++ API, which also serves as the main interface for the parser module and the basis for all other language bindings. The parser module uses the same API as external users, without any special privileges. cvc5's C++ API has been designed and written from scratch and thus is not backwards compatible with CVC4's C++ API. It is centered around the `Solver` class, which represents a cvc5 instance and implements methods for tasks such as creating terms, asserting formulas, and issuing checks.

cvc5's Python API is built on top of cvc5's C++ API using Cython [29] and makes all of cvc5's features accessible to Python users. It is a straightforward translation of the C++ API without added syntactic sugar such as operator overloading. Additionally, however, cvc5 provides a higher-level layer on top of its Python API, which is more user-friendly and *pythonic*. This layer provides automatic solver management, allows SMT terms to be constructed using Python infix operators, and converts Python objects to SMT terms of the appropriate sort. This leads to much more succinct code, as shown in Figure 2, which compares using the high- and low-level Python APIs to solve the integer equation $2 \cdot x = 4$. The higher-level Python API is based on and designed to work as a drop-in replacement for Z3py, the Python API of Z3 [90].

cvc5's Java API is implemented via the Java Native Interface (JNI), which allows Java applications to invoke native code and vice versa [83]. In contrast, CVC4 uses SWIG [28] to semi-automatically generate bindings. One of the challenges of developing a Java API, and the main motivation for implementing it manually instead of using SWIG, is the interaction between Java's garbage collector and cvc5's reference-counting mechanism for terms and sorts. The new API implements the AutoCloseable interface to destroy the underlying C++ objects in the expected order. It mostly mirrors the C++ API and supports operator overloading, iterators, and exceptions. There are a few differences from the C++ API, such as using arbitrary-precision integer pairs, specifically, pairs of Java `BigInteger` objects, to represent rational numbers. In contrast to the old Java API, the new API puts greater emphasis on using Java-native types such as `List<T>` instead of wrapper classes for C++ types such as `std::vector<T>`.

Documentation. We provide comprehensive documentation for both cvc5 users [8] and developers [6]. User documentation contains instructions for building and installing cvc5 and its dependencies, extensive documentation and examples of common uses cases for all available APIs, and a thorough description of all supported non-standard theories with examples. Developer documentation provides details of cvc5 internals and instructions for contributions, including guidelines for coding and testing, and a recommended development workflow.

Proofs. As mentioned above, cvc5 has a new proof system. Proofs are stored internally using a new custom intermediate representation. Multiple output proof formats are supported via target-specific post-processing transformations on this internal representation. The final proof object can then be pretty-printed and saved in a text file. The currently supported output proof formats include LFSC [133], Alethe [128], and the language of the Lean 4 [88] proof assistant.

CVC4 proofs exclusively used the LFSC format. cvc5 continues support for LFSC but with a new, more user-friendly syntax. LFSC is a logical framework, based on Edinburgh LF [69], which was explicitly designed to facilitate the production and checking of fine-grained proofs in SMT. It comes with a small and high-performance proof checker, which is generic in the sense that it takes as input both a proof term p and a *proof signature*, a definition of the data types and proof rules used to construct p. The checker verifies that p is well-formed with respect to the provided signature. We have defined proof signatures for all the individual theories supported by cvc5. These definitions can be combined together as needed to define a proof system for any combination of those theories. When emitting proofs in LFSC, cvc5 includes all the relevant signatures as a preamble to the proof term.

The Alethe proof format is a flexible proof format for SMT solvers based on SMT-LIB. It includes both coarse- and fine-grained steps and was first implemented in the veriT solver [34]. Alethe proofs can be checked via reconstruction within Isabelle/HOL [15, 129] as well as within Coq, the latter via the SMTCoq plugin [5, 58]. Our main motivation for producing Alethe proofs is to leverage these proof reconstruction infrastructures, thus enabling the trustworthy integration of cvc5 in Isabelle/HOL and Coq. Users of these tools can leverage the integration to dispatch selected goals to cvc5 for proving, thereby increasing the level of automation available to them without requiring a larger trusted core. These integrations represent ongoing work in cvc5 and are being carried out in close collaboration with both Isabelle/HOL and Coq experts.

Although we aim to have a similar full integration in the Lean 4 [88] proof assistant in the future, cvc5 currently only supports the use of Lean 4 as an external checker; i.e., cvc5 can emit proofs as Lean terms (for a subset of the theories supported by cvc5), and Lean 4 can then check these proofs. Since the underlying logic of Lean 4 is an extension of that of LFSC, this functionality follows an approach similar to that used for LFSC by modeling cvc5 proof rules as Lean types and reducing proof checking to type checking.

Syntax-Guided Synthesis. cvc5 has native support for syntax-guided synthesis (SyGuS) problems [3]. As mentioned, the cvc5 core has a dedicated module for encoding SyGuS problems into (higher-order) SMT formulas, annotated with syntactic restrictions. These restrictions are represented via a deep embedding into the theory of datatypes. Internally, after encoding the SyGuS problem, a sub-module of the quantifiers theory, called the synthesis engine, is the main entry point for solving. Based on the shape of the input, it uses one of three approaches. If the input problem has no syntactic restrictions, and is in *single invocation* form [114], that is, all functions to synthesize are applied to the same argument list, then it uses a quantifier-instantiation based approach. Otherwise, it uses one of two enumerative approaches, depending on the properties of the input [111]. The SyGuS solver also implements further refinements and extensions of the enumerative approaches, including algorithms for decision-tree learning [4] for programming-by-example problems, extended rewriting for enumeration [101], piecewise-independent unification [17], and static grammar-reduction techniques. Furthermore, the SyGuS solver contains specialized procedures to support an efficient implementation of interpolation and abduction.

Interpolation and Abduction. cvc5 computes abducts and Craig interpolants [51] using solvers built on top of the SyGuS solver. The solver for interpolation translates an interpolation query into a SyGuS conjecture whose solutions are interpolants. Specifically, given quantifier-free formulas A and C over any combination of the theories supported by cvc5, the interpolation solver solves for B in the SyGuS conjecture $A \rightarrow B \ \wedge \ B \rightarrow C$, with the syntactic restriction that B's free symbols range over the symbols shared by A and C. Any synthesized solution for B is, by construction, a Craig interpolant for A and C.

Abduction is the process of constructing a formula B that is enough to add to a formula A to prove some goal formula C (equivalently, to make the formula $F = A \wedge B \wedge \neg C$ unsatisfiable). cvc5's abduction solver reduces this problem to a SyGuS one where C is the formula to be synthesized and F is the semantic constraint. Optionally, the user can also impose syntactic restrictions on the abduct B. The SyGuS solver implements specific optimizations for abduction queries, such as using unsat cores to prune classes of invalid candidate solutions [110].

Non-Linear Arithmetic. The new sub-solver for non-linear arithmetic is based on cylindrical algebraic coverings and closely follows [1], with some notable extensions. The implementation uses the libpoly library [76], which provides polynomial arithmetic and most algebraic routines required for the computation of cylindrical algebraic decompositions and coverings. Infeasible subsets are computed by tracking all contributing assertions for every covering. The infeasible subset is then obtained from the union of assertions from the top-level covering. The sub-solver implements several different variable orderings, as these can have a significant impact on run-times in practice. Apart from classical variable orderings used for cylindrical algebraic decomposition, some experimental orderings based on machine learning have been implemented, roughly following ideas from England et al. [59]. (Mixed real-) integer problems are supported by dynamically injecting intervals into coverings to cover gaps that do not contain integers.

Higher-Order Logic. cvc5 has been extended with partial support for higher-order logic [18]. The extension is based on a pragmatic approach in which λ-abstractions are eliminated eagerly via lambda lifting [71]. This approach is used with the theory solver for the quantifier-free fragment of the theory of equality with uninterpreted functions (EUF) and with the quantifier-instantiation technique based on E-matching with triggers [53,89]. For the EUF solver, we added support for (dis)equality constraints between functions, via an extensionality inference rule, and for partial applications of (Curried) functions. For quantifier instantiation, we modified several of the data structures for E-matching to incorporate matching in the presence of equalities between function values, function variables, and partial function applications. The extension also uses custom axioms, such as an axiom simulating how functions are updated, to improve the generation of new λ-abstractions, since cvc5 does not yet perform HO-unification, which would allow it to synthesize arbitrary λ-abstractions.

New Bit-Vector Solver. cvc5 features a new bit-blasting solver, which supports the use of off-the-shelf SAT solvers such as CaDiCaL [31] or CryptoMiniSat [131] as SAT back-ends for *both* the eager and lazy bit-blasting approaches. In contrast, CVC4's lazy bit-blasting solver relied on a customized version of MiniSat and did not allow the use of more recent state-of-the-art SAT solvers.

Int-Blasting. In addition to bit-blasting, cvc5 implements *int-blasting* techniques, which reduce bit-vector problems to equisatisfiable non-linear integer arithmetic problems [97,138]. These techniques are orthogonal to bit-blasting and especially effective on unsatisfiable formulas over large bit-widths.

Syntax-Guided Quantifier Instantiation. cvc5 features a new theory-agnostic enumerative quantifier-instantiation technique we call *syntax-guided quantifier instantiation* [96]. This technique leverages cvc5's SyGuS solver to synthesize terms for quantifier instantiation in a counterexample-guided manner.

Unsatisfiable Cores. In cvc5, unsat (short for unsatisfiable) core extraction has been completely overhauled. It now uses the new proof infrastructure for tracking preprocessing transformations, which, differently from CVC4's, supports most of the preprocessing passes. Unsat cores can be extracted based on the constructed proof or via the tracked preprocessing and assumption-based unsat core extraction [47]. For the latter, cvc5 uses the solve-under-assumptions feature available in the MiniSat-based SAT engine. This is a lightweight solution that does not require the generation of proofs in the SAT solver and full preprocessing proofs. However, if a user requests both unsat cores and proofs, cvc5 switches to proof-based unsat core extraction using the new proof infrastructure.

Distributed and Central Policies for Equality Reasoning. As mentioned in Section 2, the Combination Engine manages theory combination, and theory solvers manage their interactions with the rest of the system via their Equality Engine. In contrast to CVC4, the policy for assigning an Equalitiy Engine to a theory solver in cvc5 is configurable. In the *distributed* policy, a new Equality Engine is generated and assigned for each theory solver. These theory solvers perform congruence closure and their theory-specific reasoning locally. The advantage of this approach is that the constraints are local to the theory and thus do not lead to overhead when combined with other theories. In the *central* policy, a single, shared Equality Engine is assigned to all theory solvers. The advantage of this approach is that communication of facts between theory solvers happens automatically, which in turn can trigger theory propagations more eagerly. Both policies use the same core Equality Engine Module. Each theory solver has been refactored to be agnostic with respect to the equality policy.

Decision Heuristic. For Boolean reasoning, in addition to MiniSat's decision heuristic, cvc5 implements a separate decision heuristic which uses the original Boolean structure of the input to keep track of the *justified* parts of the input constraints, i.e., the parts where it can infer the value of terms based on a (partial) assignment to sub-terms. To make decisions, this new heuristic traverses assertions not satisfied by the currently asserted literals, computing the desired values (starting with true as the desired value for the root) for each term until it finds an unasserted literal that would contribute towards a desired value. This heuristic is a reimplementation and extension of a heuristic [12] implemented in CVC4. The heuristic optionally prioritizes assertions that most frequently contributed to conflicts in the past using a dynamic ordering scheme.

Additional Features. Many more aspects and features have been improved and implemented with the goal of providing useful information to users and developers. Notable examples include: a complete overhaul of CVC4's mechanism for collecting statistics; improved bookkeeping for information about theory lemmas; and a general mechanism for communicating additional information to users such as quantifier instantiations and terms enumerated by the SyGuS solver.

4 Evaluation

We evaluate cvc5's overall performance (commit `5f998504`) by comparing it against Z3 4.8.12 [90] and CVC4 1.8.[7] Z3 is a widely used, high-performance SMT solver which, like cvc5, supports a wide range of theories. We compare against CVC4 to illustrate some of the performance improvements implemented as part of the move to cvc5. To run CVC4 optimally, we use the same command-line options as those in CVC4's competition script for SMT-COMP 2020 [9]. Similarly, for cvc5, we use a (slightly updated) version of the competition script from SMT-COMP 2021 [7]. For some logics, e.g., quantified logics, these scripts try multiple options in a sequential portfolio.

[7] The artifact of this evaluation is archived in the Zenodo open-access repository [14].

Division	**cvc5**	**CVC4**	**Z3**
Arith (7104)	6593	6498	**6844**
Bitvec (6045)	**5741**	5690	5664
Equality (12159)	6677	**6681**	4688
Equality+LinearArith (55948)	49395	48487	**49503**
Equality+MachineArith (4712)	**2065**	1832	1804
Equality+NonLinearArith (17260)	**11088**	10906	9341
FPArith (3170)	**2625**	2113	2593
QF Bitvec (42450)	**41569**	41448	40582
QF Equality (16254)	**16124**	16121	16115
QF Equality+Bitvec (16518)	16274	**16333**	16318
QF Equality+LinearArith (3924)	3778	3782	**3822**
QF Equality+NonLinearArith (673)	598	610	**616**
QF FPArith (76084)	**75998**	75965	75816
QF LinearIntArith (9765)	8619	**8778**	8464
QF LinearRealArith (2008)	1849	**1881**	1864
QF NonLinearIntArith (24261)	17525	16860	**18357**
QF NonLinearRealArith (11552)	**10889**	9207	10354
QF Strings (69863)	69231	**69367**	68074
Total (379750)	**346638**	342559	340819

Table 1: Benchmarks solved by cvc5, CVC4, and Z3 with a 20 minute time limit.

We ran all experiments on a cluster equipped with Intel Xeon E5-2620 v4 CPUs. We allocated one CPU core and 8GB of RAM for each solver and benchmark pair and ran each benchmark with a 20 minute time limit, the same time limit used at SMT-COMP 2021 [102]. We used all non-incremental SMT-LIB [22] benchmarks for our evaluation, with the exception of 45 (misclassified) benchmarks that have quantifiers in quantifier-free logics and 1128 (misclassified) benchmarks that have non-linear literals in linear arithmetic logics. These are known misclassifications in the current release of SMT-LIB. Note that many benchmarks in SMT-LIB come from industrial applications.

Table 1 shows the number of solved benchmarks for each solver using the same divisions as those used for SMT-COMP 2021. There were no disagreements among the solvers on the satisfiability of benchmarks. Overall, cvc5 solves the largest number of benchmarks. Compared to CVC4, cvc5 solves fewer benchmarks in the quantifier-free linear integer arithmetic division due to refactorings related to adding proof support. In the quantifier-free equality and bit-vector division, cvc5 also solves fewer benchmarks, which we attribute to the fact that the new bit-vector solver has not yet been optimized for theory combination. Finally, for quantifier-free string benchmarks, there have been bug fixes since CVC4 that affected performance.

In addition to regularly participating in SMT-COMP, cvc5 and CVC4 also participate in the CADE ATP System Competition (CASC) and in SyGuS-Comp [103]. In CASC, cvc5 tends to perform in the middle of the pack on untyped theorem divisions (unsatisfiable quantified UF in SMT-LIB parlance), and towards the top of the pack on theorems with arithmetic. The last time SyGuS-Comp was held was in 2019, when CVC4 won four out of five tracks.

CVC4 is used extensively in industry, and our users are in the process of updating to cvc5. Examples of its use include: a back-end for Zelkova, a tool developed at Amazon to reason about AWS Access Policies [10, 11, 33]; a back-end for Boogie [20], which is used in many projects including Dafny [81] and the Move Prover [137], a tool used to formally verify smart contracts; a back-end at Certora, another company engaged in formal verification of smart contracts [138]; a back-end for Sledgehammer [32], a tool for discharging proof obligations in Isabelle used by Isabelle's own industrial users; and a back-end for SPARK [70], a development environment for safety-critical Ada programs.

5 Future Work

We briefly highlight a few current development directions for cvc5.

Optimization Solver. Optimization modulo theories (OMT) [136] is an extension of SMT, which requires a solver not only to determine satisfiability but also to return a satisfying assignment (if any) that optimizes one or more objectives. OMT is already supported by several solvers including MathSAT [46] and Z3. cvc5 already has internal infrastructure for supporting OMT queries. We aim to improve and expose (through the APIs) this capability in the near future.

Theory of Bags. cvc5 has preliminary support for a theory of multisets (or *bags*) that can be implemented via a reduction to linear integer arithmetic [107]. We plan to extend this theory with higher-order combinators such as map and fold. With these combinators, and encoding relational tables as bags of tuples, cvc5 will be able to support several commonly-used table operations, with the goal of facilitating reasoning about SQL queries and database applications.

Floating-Point Arithmetic. In addition to word-blasting, we plan to leverage our work on invertibility conditions [36] to lift the local search approach for bit-vectors from [93, 94] to floating-point arithmetic.

Internal Portfolio. Due to the computational complexity of SMT, there is often no single strategy that works best for all problems. As a result, users of SMT solvers often rely on portfolio approaches to try different sets of options, either in parallel or sequentially, as we did in Section 4. Implementing portfolio approaches that use the solver as a black box is sub-optimal because some work, such as parsing, has to be duplicated. The cvc5 roadmap includes plans to support portfolio solving internally, thereby avoiding that additional overhead. We further plan to provide predefined portfolios tuned for specific use cases. As one example of the different needs of different use cases, some applications prefer the solver to always return quickly (even if the answer is "unknown") whereas others expect the solver to try as hard as possible to solve a given problem.

New Parser. cvc5's current parser is inherited from CVC4 and is based on the ANTLR 3 parser generator [105]. In addition to relying on a now deprecated version of ANTLR, the parser is unacceptably slow on large inputs and provides no API for user applications to interact with. A new parser using Flex [106] and Bison [49] is in development. The new parser will also provide an API allowing users to parse whole files or individual terms.

References

1. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Log. Algebraic Methods Program. **119**, 100633 (2021). https://doi.org/10.1016/j.jlamp.2020.100633
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), https://ieeexplore.ieee.org/document/6679385/
3. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), http://ieeexplore.ieee.org/document/6679385/
4. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 319–336 (2017). https://doi.org/10.1007/978-3-662-54577-5_18
5. 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.P., Shao, Z. (eds.) Certified Programs and Proofs. Lecture Notes in Computer Science, vol. 7086, pp. 135–150. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_12
6. cvc5 Authors: cvc5 developer documentation. https://github.com/cvc5/cvc5/wiki (2021)
7. cvc5 Authors: cvc5 SMT-COMP 2021 Single Query run script. https://github.com/cvc5/cvc5/blob/smtcomp2021/contrib/competitions/smt-comp/run-script-smtcomp-current (2021)
8. cvc5 Authors: cvc5 user documentation. https://cvc5.github.io (2021)
9. Authors, C.: CVC4 SMT-COMP 2020 Single Query run script. https://github.com/CVC4/CVC4/blob/smtcomp2020/contrib/competitions/smt-comp/run-script-smtcomp-current (2020)
10. Backes, J., Berrueco, U., Bray, T., Brim, D., Cook, B., Gacek, A., Jhala, R., Luckow, K.S., McLaughlin, S., Menon, M., Peebles, D., Pugalia, U., Rungta, N., Schlesinger, C., Schodde, A., Tanuku, A., Varming, C., Viswanathan, D.: Stratified abstraction of access control policies. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 165–176. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_9
11. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994

12. Bansal, K.: A branching heuristic in cvc4 smt solver. https://kshitij.io/articles/cvc4-branching-heuristic.pdf (2012)
13. Bansal, K., Barrett, C.W., Reynolds, A., Tinelli, C.: A new decision procedure for finite sets and cardinality constraints in SMT. CoRR **abs/1702.06259** (2017), http://arxiv.org/abs/1702.06259
14. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M.M.Y., Niemetz, A., Noetzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: Artifact for Paper cvc5: A Versatile and Industrial-Strength SMT Solver (Nov 2021). https://doi.org/10.5281/zenodo.5740365, https://doi.org/10.5281/zenodo.5740365
15. 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). https://doi.org/10.1007/s10817-018-09502-y
16. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10206, pp. 214–230 (2017). https://doi.org/10.1007/978-3-662-54580-5_13
17. Barbosa, H., Reynolds, A., Larraz, D., Tinelli, C.: Extending enumerative function synthesis via smt-driven classification. 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. 212–220. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894267
18. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 11716, pp. 35–54. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_3
19. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
20. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387. Springer (2005)
21. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011)
22. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2020), http://smt-lib.org
23. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Hermann, M., Voronkov, A. (eds.) Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '06). Lecture Notes in Computer Science, vol. 4246, pp. 512–526. Springer-Verlag (Nov 2006), phnom Penh, Cambodia

24. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. JSAT **3**(1-2), 21–46 (2007). https://doi.org/10.3233/sat190028
25. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
26. Barrett, C.W., Berezin, S.: CVC lite: A new implementation of the cooperating validity checker category B. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 3114, pp. 515–518. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_49
27. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4590, pp. 298–302. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_34
28. Beazley, D.M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: Diekhans, M., Roseman, M. (eds.) Fourth Annual USENIX Tcl/Tk Workshop 1996, Monterey, California, USA, July 10-13, 1996. USENIX Association (1996), https://www.usenix.org/legacy/publications/library/proceedings/tcl96/beazley.html
29. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: The best of both worlds. Computing in Science & Engineering **13**(2), 31–39 (2011)
30. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
31. 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., Järvisalo, 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)
32. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6803, pp. 116–130. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_11
33. Bouchet, M., Cook, B., Cutler, B., Druzkina, A., Gacek, A., Hadarean, L., Jhala, R., Marshall, B., Peebles, D., Rungta, N., Schlesinger, C., Stephens, C., Varming, C., Warfield, A.: Block public access: trust safety verification of access control policies. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 281–291. ACM (2020). https://doi.org/10.1145/3368089.3409728
34. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 5663, pp. 151–156. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_12
35. 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 Deduc-

tion - 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

36. Brain, M., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Invertibility conditions for floating-point formulas. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 116–136. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_8
37. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: TACAS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. LNCS, vol. 11427, pp. 79–98. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_5
38. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA. pp. 69–76. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351141
39. Bromberger, M., Weidenbach, C.: Fast cube tests for LIA constraint solving. In: IJCAR. Lecture Notes in Computer Science, vol. 9706, pp. 116–132. Springer (2016)
40. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
41. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. 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. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
42. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 510–517. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_29
43. Christ, J., Hoenicke, J.: Weakly equivalent arrays. In: FroCos. Lecture Notes in Computer Science, vol. 9322, pp. 119–134. Springer (2015)
44. 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
45. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS

2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 58–75 (2017). https://doi.org/10.1007/978-3-662-54577-5_4

46. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013)
47. Cimatti, A., Griggio, A., Sebastiani, R.: Computing small unsatisfiable cores in satisfiability modulo theories. J. Artif. Intell. Res. (JAIR) **40**, 701–728 (2011). https://doi.org/10.1613/jair.3196
48. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) 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. 38–47. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
49. Corbett, R.: Gnu bison (2021), https://www.gnu.org/software/bison/
50. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 360–368. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_26
51. Craig, W.: Linear reasoning. A new form of the herbrand-gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
52. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c - A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
53. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
54. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. Formal Methods Syst. Des. **39**(3), 246–260 (2011)
55. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
56. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: CAV. Lecture Notes in Computer Science, vol. 4144, pp. 81–94. Springer (2006)
57. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003)
58. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.W.: Smtcoq: A plug-in for integrating SMT solvers into coq. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10427, pp. 126–133. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_7
59. England, M., Bradford, R.J., Davenport, J.H., Wilson, D.J.: Choosing a variable ordering for truth-table invariant cylindrical algebraic decomposition by incremental triangular decomposition. In: Hong, H., Yap, C. (eds.) Mathematical

Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8592, pp. 450–457. Springer (2014). https://doi.org/10.1007/978-3-662-44199-2_68

60. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) 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. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
61. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 519–531. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_52
62. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). https://doi.org/10.1145/2093548.2093564
63. Griggio, A.: An Effective SMT Engine for Formal Verification. Ph.D. thesis, University of Trento, Italy (2009)
64. Griggio, A.: A practical approach to satisfiability modulo linear integer arithmetic. Journal on Satisfiability, Boolean Modeling and Computation **8**(1-2), 1–27 (2012)
65. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 62–73. ACM (2011). https://doi.org/10.1145/1993498.1993506
66. Hadarean, L.: An efficient and trustworthy theory solver for bit-vectors in satisfiability modulo theories. Ph.D. thesis, Citeseer (2015)
67. Hadarean, L., Barrett, C.W., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 340–355. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_24
68. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12031, pp. 161–179. Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_11
69. Harper, R., Honsell, F., Plotkin, G.: A Framework for Defining Logics. Journal of the Association for Computing Machinery **40**(1), 143–184 (Jan 1993)
70. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from proof failures in SPARK. In: Nicola, R.D., eva Kühn (eds.) Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9763, pp. 215–233. Springer (2016). https://doi.org/10.1007/978-3-319-41591-8_15
71. Hughes, R.J.M.: Super combinators: a new implementation method for applicative languages. In: Symposium on LISP and Functional Programming. pp. 1–10 (1982)

72. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 547–553. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_35
73. Iosif, R., Serban, C., Reynolds, A., Sighireanu, M.: Encoding separation logic in smt-lib v2.5 (2018)
74. Jovanovic, D., Barrett, C.W.: Polite theories revisited. In: LPAR (Yogyakarta). Lecture Notes in Computer Science, vol. 6397, pp. 402–416. Springer (2010)
75. Jovanovic, D., Barrett, C.W.: Being careful about theory combination. Formal Methods Syst. Des. **42**(1), 67–90 (2013)
76. Jovanovic, D., Dutertre, B.: Libpoly: A library for reasoning about polynomials. In: Brain, M., Hadarean, L. (eds.) Proceedings of the 15th International Workshop on Satisfiability Modulo Theories affiliated with the International Conference on Computer-Aided Verification (CAV 2017), Heidelberg, Germany, July 22 - 23, 2017. CEUR Workshop Proceedings, vol. 1889, pp. 28–39. CEUR-WS.org (2017), http://ceur-ws.org/Vol-1889/paper3.pdf
77. Katz, G., Barrett, C.W., 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 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 93–100. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886666
78. King, T.: Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic. Ph.D. thesis, New York University (2014)
79. King, T., Barrett, C.W., Dutertre, B.: Simplex with sum of infeasibilities for SMT. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 189–196. IEEE (2013), https://ieeexplore.ieee.org/document/6679409/
80. King, T., Barrett, C.W., Tinelli, C.: Leveraging linear and mixed integer programming for SMT. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014. pp. 139–146. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987606
81. Leino, K.M.: Accessible software verification with dafny. IEEE Software **34**(06), 94–97 (nov 2017). https://doi.org/10.1109/MS.2017.4121212
82. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
83. Liang, S.: The Java Native interface : programmer's guide and specification / Sheng Liang. Java series, Addison-Wesley, Reading, Mass. ; Harlow, England (1999)
84. Liang, T., Reynolds, A., Tinelli, C., Barrett, C.W., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 646–662. Springer (2014)
85. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.W.: A decision procedure for regular membership and length constraints over unbounded strings.

In: FroCos. Lecture Notes in Computer Science, vol. 9322, pp. 135–150. Springer (2015)

86. Mattarei, C., Mann, M., Barrett, C.W., Daly, R.G., Huff, D., Hanrahan, P.: Cosa: Integrated verification for agile hardware design. In: Bjørner, N., 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.2018.8603014
87. Meng, B., Reynolds, A., Tinelli, C., Barrett, C.W.: Relational constraint solving in SMT. 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. 148–165. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_10
88. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
89. de Moura, L.M., Bjørner, N.: Efficient e-matching for SMT solvers. In: Pfenning, F. (ed.) Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_13
90. 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
91. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD. pp. 45–52. IEEE (2009)
92. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621
93. Niemetz, A., Preiner, M.: Ternary propagation-based local search for more bit-precise reasoning. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 214–224. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29
94. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. Formal Methods Syst. Des. **51**(3), 608–636 (2017). https://doi.org/10.1007/s10703-017-0295-6
95. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: On solving quantified bit-vector constraints using invertibility conditions. Formal Methods Syst. Des. **57**(1), 87–115 (2021). https://doi.org/10.1007/s10703-020-00359-9
96. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Syntax-guided quantifier instantiation. 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. 145–163. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_8

97. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C.W., Tinelli, C.: Towards bit-width-independent proofs in SMT solvers. In: Fontaine, P. (ed.) Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11716, pp. 366–384. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_22
98. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , btormc and boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) 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. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
99. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). J. ACM **53**(6), 937–977 (2006). https://doi.org/10.1145/1217856.1217859
100. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
101. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C.W., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. 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. 279–297. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_20
102. Organizers, S.C.: SMT-COMP 2021. https://smt-comp.github.io/2021/ (2021)
103. Organizers, S.C.: SyGuS-Comp 2019. https://sygus.org/comp/2019/ (2021)
104. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The sygus language standard version 2.1 (2021)
105. Parr, T.: ANTLRv3 (2021), https://www.antlr3.org/
106. Paxson, V.: Flex lexical analyser generator (2021), https://github.com/westes/flex
107. Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings. Lecture Notes in Computer Science, vol. 4905, pp. 218–232. Springer (2008). https://doi.org/10.1007/978-3-540-78163-9_20
108. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: FroCoS. Lecture Notes in Computer Science, vol. 3717, pp. 48–64. Springer (2005)
109. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 112–131. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_7
110. Reynolds, A., Barbosa, H., Larraz, D., Tinelli, C.: Scalable algorithms for abduction via enumerative syntax-guided synthesis. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Con-

ference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 141–160. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_9

111. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 74–83. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_5
112. Reynolds, A., Blanchette, J.C.: A decision procedure for (co)datatypes in SMT solvers. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195, pp. 197–213. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_13
113. Reynolds, A., Blanchette, J.C., Cruanes, S., Tinelli, C.: Model finding for recursive functions in SMT. In: Olivetti, N., Tiwari, A. (eds.) Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9706, pp. 133–151. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_10
114. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. 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 II. Lecture Notes in Computer Science, vol. 9207, pp. 198–216. Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_12
115. Reynolds, A., Iosif, R., Serban, C., King, T.: A decision procedure for separation logic in SMT. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 244–261 (2016). https://doi.org/10.1007/978-3-319-46520-3_16
116. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Formal Methods Syst. Des. **51**(3), 500–532 (2017). https://doi.org/10.1007/s10703-017-0290-y
117. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8931, pp. 80–98. Springer (2015). https://doi.org/10.1007/978-3-662-46081-8_5
118. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: CAV (2). Lecture Notes in Computer Science, vol. 11562, pp. 23–42. Springer (2019)
119. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: A decision procedure for string to code point conversion. In: IJCAR (1). Lecture Notes in Computer Science, vol. 12166, pp. 218–237. Springer (2020)
120. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: Reductions for strings and regular expressions revisited. In: FMCAD. pp. 225–235. IEEE (2020)
121. Reynolds, A., Tinelli, C., Goel, A., Krstic, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 640–655. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_42

122. Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., Barrett, C.W.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 377–391. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_26
123. Reynolds, A., Tinelli, C., Jovanovic, D., Barrett, C.W.: Designing theory solvers with extensions. In: Dixon, C., Finger, M. (eds.) Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10483, pp. 22–40. Springer (2017). https://doi.org/10.1007/978-3-319-66167-4_2
124. Reynolds, A., Tinelli, C., de Moura, L.M.: Finding conflicting instances of quantified formulas in SMT. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014. pp. 195–202. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987613
125. Reynolds, A., Viswanathan, A., Barbosa, H., Tinelli, C., Barrett, C.W.: Datatypes with shared selectors. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10900, pp. 591–608. Springer (2018). https://doi.org/10.1007/978-3-319-94205-6_39
126. Reynolds, A., Woo, M., Barrett, C.W., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: CAV (2). Lecture Notes in Computer Science, vol. 10427, pp. 453–474. Springer (2017)
127. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. Commun. ACM **59**(2), 114–122 (2016). https://doi.org/10.1145/2863701
128. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). In: Keller, C., Fleury, M. (eds.) Workshop on Proof eXchange for Theorem Proving (PxTP). EPTCS, vol. 336, pp. 49–54 (2021). https://doi.org/10.4204/EPTCS.336.6, https://doi.org/10.4204/EPTCS.336.6
129. 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). https://doi.org/10.1007/978-3-030-79876-5_26
130. Sheng, Y., Zohar, Y., Ringeissen, C., Lange, J., Fontaine, P., Barrett, C.W.: Politeness for the theory of algebraic datatypes. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 238–255. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_14
131. Soos, M.: CryptoMiniSat. https://github.com/msoos/cryptominisat (2020)
132. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A cooperating validity checker. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2404, pp. 500–504. Springer (2002). https://doi.org/10.1007/3-540-45657-0_40
133. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design **42**(1), 91–118 (2013). https://doi.org/10.1007/s10703-012-0163-3
134. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)

135. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: Beckert, B., Hähnle, R. (eds.) Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer (2008). https://doi.org/10.1007/978-3-540-79124-9_10
136. Trentin, P.: Optimization Modulo Theories with OptiMathSAT. Ph.D. thesis, University of Trento (2019)
137. Zhong, J.E., Cheang, K., Qadeer, S., Grieskamp, W., Blackshear, S., Park, J., Zohar, Y., Barrett, C.W., Dill, D.L.: The move prover. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 137–150. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_7
138. Zohar, Y., Irfan, A., Mann, M., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-Precise Reasoning via Int-Blasting, to appear in the proceedings of VMCAI 2022

Clausal Proofs for Pseudo-Boolean Reasoning*

Randal E. Bryant[1] ✉, Armin Biere[2], and Marijn J. H. Heule[1]

[1] Carnegie Mellon University, Pittsburgh, PA, United States
{Randy.Bryant, mheule}@cs.cmu.edu
[2] Albert-Ludwigs University, Freiburg, Germany
biere@cs.uni-freiburg.de

Abstract. When augmented with a Pseudo-Boolean (PB) solver, a Boolean satisfiability (SAT) solver can apply apply powerful reasoning methods to determine when a set of parity or cardinality constraints, extracted from the clauses of the input formula, has no solution. By converting the intermediate constraints generated by the PB solver into ordered binary decision diagrams (BDDs), a proof-generating, BDD-based SAT solver can then produce a clausal proof that the input formula is unsatisfiable. Working together, the two solvers can generate proofs of unsatisfiability for problems that are intractable for other proof-generating SAT solvers. The PB solver can, at times, detect that the proof can exploit modular arithmetic to give smaller BDD representations and therefore shorter proofs.

1 Introduction

Like all complex software, modern satisfiability (SAT) solvers are prone to bugs. In seeking to maximize their performance, developers may attempt optimizations that are either unsound or incorrectly implemented. Requiring a solver to be formally verified is not feasible for current solvers. On the other hand, ensuring that each execution of the solver yields the correct result has become a standard requirement. For a satisfiable formula, the solver can generate a purported solution, and this can be checked directly. For an unsatisfiable formula, the solver can produce a proof of unsatisfiability in a logical framework that enables checking by an efficient and trusted proof checker. Proof generation is a vital capability when SAT solvers are used for formal correctness and security verification, and for mathematical theorem proving.

Most high-performance, proof-generating SAT solvers are based on conflict-driven, clause-learning (CDCL) algorithms [42]. Although the methods used by earlier solvers were limited to steps that could be justified within a resolution framework [43, 52], modern solvers employ a variety of optimizations that require a more expressive proof framework, with the most common being Deletion Resolution Asymmetric Tautology (DRAT) [31,50]. Like resolution proofs, a DRAT proof is a *clausal proof* consisting of a sequence of clauses, each of which preserves the satisfiability of the preceding clauses. An unsatisfiable proof starts with the clauses of the input formula and ends with an empty clause, indicating logical falsehood. The fact that this clause can be derived from the original formula proves that the original formula cannot be satisfied.

* The first and third authors were supported by the U. S. National Science Foundation under grant CCF-2108521

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 443–461, 2022.
https://doi.org/10.1007/978-3-030-99524-9_25

Even with the capabilities of the DRAT framework, some solvers employ reasoning techniques for which they cannot generate unsatisfiability proofs. A number of SAT solvers can extract parity constraints from the input clauses and solve these as linear equations over the integers modulo 2 [6, 30, 37, 47]. Some can also detect and reason about cardinality constraints [6]. However, all these programs revert to standard CDCL when proof generation is required. To overcome the proof-generating limitations of current solvers, some have suggested using more powerful proof frameworks, for example, based on pseudo-Boolean constraints [27] or Binary Decision Diagrams [5]. Staying with DRAT avoids the need to develop, certify, and deploy new proof systems, file formats, and checkers.

Current CDCL solvers do not use the full power of the DRAT framework. In particular, DRAT supports adding *extension variables* to a clausal proof, in the style of extended resolution [48]. These variables serve as abbreviations for formulas over existing input and extension variables. Compared to standard resolution, allowing extension variables can yield proofs that are exponentially more compact [19], and the same holds for the extension rule in DRAT. In general, however, CDCL solvers have been unable to exploit this capability, with the exception that some of their preprocessing and inprocessing techniques [8, 34] require extension variables [39]. One solver attempted to introduce extension variables as it operated [3], but it achieved only modest success.

In 2006, Biere, Jussila, and Sinz demonstrated that the underlying logic behind algorithms for constructing Reduced, Ordered Binary Decision Diagrams (BDDs) [10] can be encoded as steps in an extended resolution framework [35, 46]. By introducing an extension variable for each BDD node generated, the logic for each recursive step of standard BDD operations can be expressed with a short sequence of proof steps. BDDs provide a systematic way to exploit the power of extension variables. The recently developed solver PGBDD [11, 12] (for "proof-generating BDD") builds on this work with a more general capability for existentially quantifying variables. It can generate unsatisfiability proofs for several classic challenge problems for which the shortest possible standard resolution proofs are of exponential size.

We show that BDDs can provide a bridge between *pseudo-Boolean reasoning* and clausal proofs. Pseudo-Boolean (PB) constraints have the form $\sum_{j=1,n} a_j \, x_j \rhd b$, where each variable x_j can be assigned value 0 or 1, the coefficients a_j and constant b are integers, and the relation symbol $\rhd$ is either $=$, $\geq$, or $\equiv \bmod r$ for some modulus r. Both parity and cardinality constraints can be expressed as PB constraints. A PB solver can employ Gaussian elimination or Fourier-Motzkin elimination [21, 51] to determine when a set of constraints is unsatisfiable. Our newly developed program PGPBS (for "proof-generating pseudo-Boolean solver") augments PGBDD with a pseudo-Boolean solver, combining the power of PB reasoning with DRAT proof generation.

To enable proof generation, the PB solver generates BDD representations of its intermediate constraints and has proof-generating BDD operations construct proofs that each of these constraints is logically implied by previous constraints. When the PB solver reaches a constraint that cannot be satisfied, e.g., the equation $0 = 2$, the constraint will be represented by the *false* BDD leaf $\bot$, which yields a proof step consisting of the empty clause. The resulting proof is checkable within the DRAT framework without any reference to pseudo-Boolean constraints or BDDs. Barnett and Biere [5] also proposed using BDDs when proving that the constraints generated by a PB solver were logically implied by their predecessors, but they proposed doing so in a separate proof framework rather than as the solver operates.

As an optimization, the PB solver can automatically detect cases where the unsatisfiability proof for an integer-constraint problem can use modular arithmetic. This leads to more compact BDD representations, and therefore shorter proofs.

We demonstrate the power of PGPBS's combination of BDDs and pseudo-Boolean reasoning by showing that that it can achieve polynomial scaling on two classes of problems for which CDCL solvers have exponential performance. These include parity constraints involving exclusive-or operations [17, 49] and cardinality constraints, including the mutilated chessboard [2] and pigeonhole problems [29]. Although PGBDD on its own can also achieve polynomial scaling for both classes of problems, incorporating pseudo-Boolean reasoning makes the solver much more robust. It can handle wider variations in the problem definition, how the problem is encoded as clauses, and the BDD variable ordering. It also operates with greater automation, requiring no guidance or hints from the user. These capabilities eliminate major shortcomings of PGBDD.

2 Pseudo-Boolean Constraints

Let x_j, for $1 \leq j \leq n$, be a set of variables, each of which may be assigned value 0 or 1, and a_j, for $1 \leq j \leq n$, be a set of integer coefficients. Constant b is also an integer. A *pseudo-Boolean* constraint is of the form $\sum_{j=1,n} a_j\, x_j \rhd b$, with $\rhd$ defining the relation between the left-hand weighted sum and the right-hand constant. For an *integer equation*, $\rhd$ is $=$, i.e., the two sides must be equal. For an *ordering constraint*, $\rhd$ is $\geq$. For a *modular equation*, $\rhd$ is $\equiv \bmod r$, where r is the chosen modulus.

Three constraint types are of special importance for solving cardinality problems. An *at-least-one* (ALO) constraint is an ordering constraint with $a_j \in \{0, +1\}$ for all j, and $b = +1$. An *at-most-one* (AMO) constraint is an ordering constraint with $a_j \in \{-1, 0\}$ for all j, and $b = -1$. An *exactly-one* constraint is an integer equation with $a_j \in \{0, +1\}$ for all j and $b = +1$.

2.1 BDD Representations

Many researchers have investigated the use of BDDs to represent pseudo-Boolean constraints [1,24,33]. As examples, Figure 1 shows BDD representations of the three forms of constraints for $n = 10$ and $b = 0$, with $a_j = +1$ for odd values of j and -1 for even values. The modular equation has $r = 3$. The BDDs for both the integer equation (A) and ordering constraint (B) have an increasing number of nodes at each level for the first $n/2$ levels, with a node at level k for each possible value of the *prefix sum* $\sum_{j=1,k-1} a_j\, x_j$. As the level k approaches n, however, the number of nodes at each level decreases. If a prefix sum becomes too extreme on the negative side, it becomes impossible for the remaining values to cause the sum to reach $b = 0$. For the integer equation, a similar phenomenon happens if a prefix sum becomes too extreme on the positive side. For an ordering constraint, a sufficiently positive prefix sum will guarantee that the total sum will be at least 0. For the modular sum (C), the number of nodes at any level cannot exceed r—one for each possible value of the prefix sum modulo r.

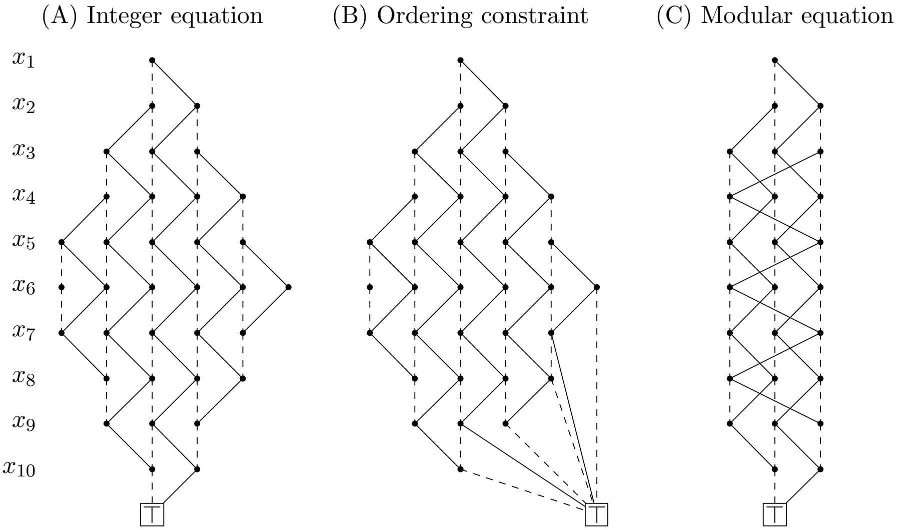


Fig. 1. Example BDD representations of pseudo-Boolean equations and ordering constraints. Solid (respectively, dashed) lines indicate the branch when the variable is assigned 1 (resp., 0). The leaf representing the *false* Boolean constant $\perp$ and its incoming edges are omitted.

Letting $a_{\max} = \max_{1 \leq j \leq n} |a_j|$, the BDD representation of an integer equation or ordering constraint will have at most $2\,a_{\max} \cdot n$ nodes at any level, while the representation of a modular equation will have at most r nodes at any level. Although large values of $a_{\max}$ ($a_{\max} \gg n$), can cause the BDDs to be of exponential size [1,33], our use of them will assume that both $a_{\max}$ and r are small constants. The BDD representations will then be $O(n^2)$ for integer equations and ordering constraints, and $O(n)$ for modular equations. These bounds are independent of the BDD variable ordering.

Most BDD operations are implemented via the *Apply* algorithm [10], recursively traversing a set of argument BDDs to either construct a new BDD or to test some property of existing ones. The BDDs representing pseudo-Boolean constraints are *levelized*: every branch from a node at level j goes to a leaf node or to a node at level $j + 1$. We can therefore derive a bound on the maximum number of recursive steps to perform an operation on k argument BDDs, assuming both $a_{\max}$ and r are small constants. Due to the caching of intermediate results, the maximum number of steps at each level will be bounded by the product of the number of argument nodes at this level. The operation will therefore have worst-case complexity $O(n^{k+1})$ for integer equations and ordering constraints, while it will have complexity $O(k \cdot n)$ for modular equations.

2.2 Solving Systems of Equations with Gaussian Elimination

We use a formulation of Gaussian elimination that scales each derived equation, rather than dividing by the pivot value [4,44]. Performing the steps therefore requires only addition and multiplication. This allows maintaining integer coefficients and automatically detecting a minimum, possibly non-prime, modulus for equation solving.

Consider a system of integer or modular equations E, where each equation $\mathbf{e}_i \in E$, is of the form $\sum_{j=1,n} a_{i,j}\, x_j = b_i$. Applying one step of Gaussian elimination involves selecting a *pivot*, consisting of an equation $\mathbf{e}_s \in E$ and a variable x_t such that $a_{s,t} \neq 0$. Then an equation $\mathbf{e}'_i$ is generated for each value of i:

$$\mathbf{e}'_i = \begin{cases} \mathbf{e}_i & a_{i,t} = 0 \\ -a_{i,t} \cdot \mathbf{e}_s + a_{s,t} \cdot \mathbf{e}_i, & a_{i,t} \neq 0 \end{cases} \tag{1}$$

where operations $+$ and $\cdot$ denote addition and scalar multiplication of equations. Observe that $a'_{i,t} = 0$ for all equations $\mathbf{e}'_i$. Letting $E \leftarrow \{\mathbf{e}'_i | i \neq s\}$, this step has reduced both the number of equations in E and the number of variables in the equations by one.

Repeated applications of the elimination step will terminate when either 1) all equations have been eliminated, or 2) an unsolvable equation is encountered. For case 1, the system has solutions, but these may, in general, assign values other than 0 and 1 to the variables. (Importantly, parity constraints are represented by modular equations with $r = 2$. Their solutions *will* be 0-1 valued, and so a SAT solver can make use of them [30, 37].) For case 2, if some elimination step generates an equation of the form $0 = b$ with $b \neq 0$, then this equation has no solution in any case, and therefore neither did the original system. Our proofs of unsatisfiability rely on reaching this condition.

For the modular case, all coefficients and the constants are kept within the range 0 to $r - 1$. For integer equations, the coefficients can grow exponentially in m. Fortunately, the cardinality problems we consider only require coefficient values -1, 0, and $+1$.

As we have seen, the BDD representations of modular equations have bounded width, making them both more compact and making the algorithms that operate on them more efficient than for integer equations. As we will see, the unsatisfiability proof generated by applying Gaussian elimination to a system of modular equations can be significantly more compact than for the same equations over integers. This gives rise to an optimization we call *modulus auto-detection.* The idea is to apply Gaussian elimination to a set of integer equations, recording the dependencies between the equations generated, but without performing any proof generation. Once the solver reaches an equation of the form $0 = b$ where $b \neq 0$, it chooses the smallest $r \geq 2$ such that $b \bmod r \neq 0$. It then generates a proof, reinterpreting the Gaussian elimination steps using modulo-r arithmetic. Since the only operations of (1) are multiplication and addition, the final equation will be $0 \equiv b \pmod{r}$, which has no solution. Here we can see that allowing r to be composite is both valid and may be optimal. For example, the smallest choice for $b = 30$ would be $r = 4$, rather than the prime $r = 7$. Auto-detection can be applied whenever Gaussian elimination encounters an unsolvable equation.

2.3 Solving Systems of Ordering Constraints with Fourier-Motzkin Elimination

Consider a set C, consisting of constraints $\mathbf{c}_i$ of the form $\sum_{j=1,n} a_{i,j}\, x_j \geq b_i$. Applying one step of Fourier-Motzin elimination [21, 51] to this system involves identifying a *pivot*, consisting of a variable x_t such that $a_{k,t} \neq 0$ for at least one value of k. The set is partitioned into three sets by assigning each constraint $\mathbf{c}_i$ to C^+, C^-, or C^0, depending on whether coefficient $a_{i,t}$ is positive, negative, or zero, respectively. For each pair i and i' such that $\mathbf{c}_i \in C^+$ and $\mathbf{c}_{i'} \in C^-$, a new constraint $\mathbf{c}_{i,i'}$ is generated as:

$$\mathbf{c}_{i,i'} = -a_{i',t} \cdot \mathbf{c}_i + a_{i,t} \cdot \mathbf{c}_{i'} \tag{2}$$

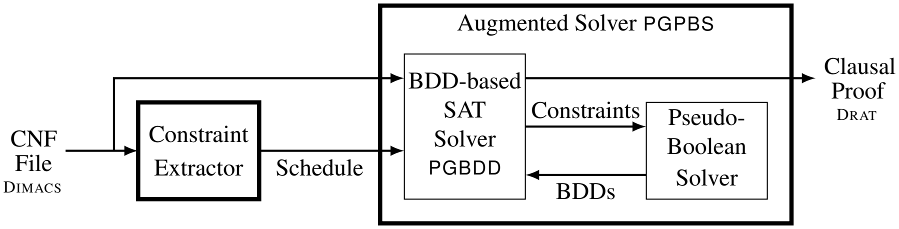


Fig. 2. Overall Structure of PGPBS. It augments the BDD-based SAT solver PGBDD with inferences from a pseudo-Boolean constraint solver. The constraint extractor is a separate program.

(Note that the multiplication is always by positive values, since $a_{i',t} < 0$.) Letting $C \leftarrow C^0 \cup \{\mathbf{c}_{i,i'} \,|\, \mathbf{c}_i \in C^+, \mathbf{c}_{i'} \in C^-\}$, all of these constraints have coefficient 0 for variable x_t. Therefore this step has reduced the number of variables in the constraints by one, but it may have increased the number of constraints.

As with Gaussian elimination, repeated application of the elimination step will terminate when either 1) all variables have been eliminated or 2) an unsolvable constraint is encountered. With case 1, the constraints can be satisfied, although possibly by assigning values other than 0 or 1 to some of the variables. An unsolvable constraint (case 2) is one where the sum of the positive coefficients is less than the constant term. If such a constraint is encountered, then the original system of constraints has no solution.

Fourier-Motzkin elimination would appear to be hopelessly inefficient. The number of constraints can grow exponentially as the elimination proceeds, and the coefficients can grow doubly exponentially. Fortunately, the cardinality problems we consider have the property that for any variable x_t, there is at most one constraint $\mathbf{c}_i$ having $a_{i,t} = +1$, at most constraint $\mathbf{c}_{i'}$ having $a_{i',t} = -1$, and no other constraint with a non-zero coefficient at position t. This property is maintained by each elimination step, and so the number of constraints will decrease with each step, and the coefficients will be restricted to the values -1, 0, and $+1$.

3 Overall Operation

Figure 2 illustrates the program structure. The pair of programs—extractor and solver—supports the standard flow for proof-generating SAT solvers, reading the input conjunctive normal form (CNF) formula expressed in the standard DIMACS format and generating proofs in the standard DRAT format. No other guidance or hint is provided. The constraint extractor identifies pseudo-Boolean constraints encoded as clauses in the input file and generates a *schedule* indicating how clauses should be combined and quantified to derive BDD representations of the constraints. PGPBS augments the SAT solver PGBDD with a PB solver. PGBDD supplies the constraints to the PB solver, which applies either Gaussian elimination or Fourier-Motzkin elimination. The PB solver generates BDD representations of the constraints it generates, and, since the BDD library generates proof steps while performing BDD operations, it can generate a proof that each new constraint is logically implied by previous constraints. When the PB solver encounters an unsolvable constraint, an empty clause is generated, completing the proof.

(A)
Exclusive-Or/Nor

CLAUSES

```
-1  2  6 0
 1 -2  6 0
 1  2 -6 0
-1 -2 -6 0
 3  6  7 0
-3 -6  7 0
-3  6 -7 0
 3 -6 -7 0
```

SCHEDULE

```
c 1 2 3 4
a 3
=2 0 1.1 1.2 1.6
c 5 6 7 8
a 3
=2 1 1.3 1.6 1.7
```

(B)
Exactly-one, direct encoding

CLAUSES

```
1 2 3 4 0
-1 -2 0
-1 -3 0
-1 -4 0
-2 -3 0
-2 -4 0
-3 -4 0
```

SCHEDULE

```
c 1 2 3 4 5 6 7
a 6
= 1 1.1 1.2 1.3 1.4
```

(C)
At-most-one, Sinz encoding [45]

CLAUSES

```
-1  5 0
-2  5 0
-1 -2 0
-5 -3 0
-5  6 0
-3  6 0
-6 -4 0
```

SCHEDULE

```
c 1 2 4 5
a 3
q 5
c 6 7
a 2
q 6
c 3
a 1
>= -1 -1.1 -1.2 -1.3 -1.4
```

Fig. 3. Examples of pseudo-Boolean constraints extracted from CNF representations. Schedules use a stack notation indicating clauses, conjunction and quantification operations, and constraints.

3.1 Constraint Extraction

The constraint extractor uses heuristic methods to identify how the input clauses match standard patterns for exclusive-or/nor, ALO, and AMO constraints. The heuristics are independent of any ordering of the clauses or variables, although they do depend on the polarities of the literals. The generated schedule indicates how to combine clauses and to quantify variables to give the different constraints. The schedule uses a stack notation, having the following commands:

c $c_1, \ldots, c_k$	Generate and push the BDDs for the specified clauses.
a m	Pop the top $m+1$ elements. Combine with m AND operations. Push the result.
q $v_1, \ldots, v_k$	Quantify the top element by the specified variables.
C $b\ a_1 . v_1, \ldots, a_k . v_k$	Confirm that the top stack element implies the constraint

The different constraint types C are '=' for integer equations, '=2' for mod-2 equations, and '>=' for integer orderings. Each constraint line lists the constant b and then indicates the non-zero terms as a combination of coefficient and variable, separated by '.'.

Figure 3 provides a series of examples illustrating the operation of the extractor. A k-way exclusive-or or exclusive-nor (A) is encoded with 2^{k-1} clauses (here $k = 3$), listing all combinations of the negated variables having even (XOR) or odd (XNOR) parity. The schedule lists the clause numbers, forms their conjunction, and indicates a mod-2 equation. The constant b is 1 for exclusive-or and 0 for exclusive-nor.

An exactly-one constraint (B) can be expressed as a combination of an ALO constraint and an AMO constraint. The extractor assumes that any clause with all literals having positive polarity encodes an ALO constraint. In this example, a k-way AMO constraint ($k = 4$) is encoded directly as a set of $k\,(k-1)/2$ binary clauses.

An AMO constraint can be also encoded with auxiliary variables (B) in variety of ways, including that devised by Sinz [45]. The extractor examines how variables occur in binary clauses. Those that occur only with negative polarity are assumed to be constraint variables, while those that have mixed polarity are assumed to be auxiliary variables. As is shown, the generated schedule for an AMO constraint encoded with auxiliary variables employs *early quantification* [13] to linearize the conjuncting of clauses and the quantification of auxiliary variables.

The heuristics used for identifying auxiliary variables and partitioning the clauses into distinct constraints apply to a wide range of AMO constraints, including those using hierarchical encodings [16, 36] and those considered in other constraint extraction programs [9]. Our method can be overly optimistic, labeling some subsets of clauses incorrectly. Fortunately, any such error will be quickly identified when the solver attempts to prove that the BDD generated by conjuncting the clauses and quantifying the auxiliary variables implies the BDD generated for the constraint.

3.2 Solver Operation

The SAT solver portion of PGPBS can generate BDD representations of input clauses and perform conjunction and existential quantification operations on BDDs [11, 12]. As the solver manipulates BDDs to track the solution state, it also generates clauses according to resolution and extension proof rules. The state of the solver at any time is captured by a set of *terms* $T_1, T_2, \ldots, T_n$, where each term T_i consists of:

- A root node u_i in the BDD.
- The extension variable associated with this node, also written as u_i.
- A unit clause, included in the proof clauses, consisting of extension variable u_i, asserting that the Boolean function represented by BDD node u_i evaluates to true for any variable assignment that satisfies the input clauses.
- Implicitly, the set $\theta(u_i)$ of all *defining clauses* that were added to the proof when introducing the extension variables for the nodes in the BDD subgraph having root u_i. These provide the semantic model for the BDD within the proof framework.

The BDD package supports proof-generating BDD operations ApplyAnd, used to perform conjunction, and ProveImplication, used to generate proofs of implication. The ApplyAnd operation takes as arguments BDD roots u and v, and it generates a BDD representation with root w of their conjunction. It also generates a proof of the clause $\overline{u} \vee \overline{v} \vee w$, proving the implication $u \wedge v \rightarrow w$. The ProveImplication operation performs implication testing without generating any new BDD nodes. It takes as arguments BDD roots u and v, and it generates a proof of the clause $\overline{u} \vee v$, proving that $u \rightarrow v$. An error is signaled if the implication does not hold.

When the solver encounters a clause command in the schedule file, it generates a term T_i for each of the specified input clauses C_i and pushes the term onto a stack. It also generates the proof $\theta(u_i), C_i \vdash u_i$, i.e., that function represented by BDD node u_i will evaluate to true for any variable assignment that satisfies the clause.

When the solver encounters a conjunction or quantification command, it creates a new term by performing the specified operation and proving that it is implied by earlier terms. Given newly generated BDD root u_{n+1}, it must prove that u_{n+1} is *implication redundant* with respect to the existing terms. That is, if u_{n+1} was generated by applying some operation to terms $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$, then it must generate a proof of the clause $\overline{u}_{i_1} \vee \overline{u}_{i_2} \vee \cdots \vee \overline{u}_{i_k} \vee u_{n+1}$. This clause can then be resolved with the unit clauses associated with the existing terms to yield the unit clause u_{n+1}, allowing a new term T_{n+1} to be added. If some step generates a term T_{n+1} with BDD representation $u_{n+1} = \bot$, it will also generate the empty clause, completing a proof of unsatisfiability.

The PB solver portion of PGPBS can generate BDD representations of the intermediate constraints it creates. The SAT solver generates a new term for each of these BDDs. The proof generator need not have any understanding of the operation of the PB solver, and vice-versa. Suppose some set of input clauses encodes a pseudo-Boolean constraint, possibly using auxiliary variables, as was illustrated in Figure 3. The SAT solver performs the series of conjunction and quantification operations specified by the schedule to reduce the clauses to a single term T_n consisting of BDD root u_n and unit clause u_n. The auxiliary variables have been quantified away, and so u_n depends only on the constraint variables. It passes the constraint to the PB solver, which generates its BDD representation with root u_{n+1}. The SAT solver uses the PROVEIMPLICATION operation to generate the clause $\overline{u}_n \vee u_{n+1}$. This can be resolved with unit clause u_n to generate the unit clause u_{n+1}, and so the BDD representation of the constraint becomes term T_{n+1}. (Typically, the two BDDs are identical and so the implication holds trivially.) This process is repeated to convert the input formula into a set of pseudo-Boolean constraints, each represented as a term in the SAT solver.

Once the SAT solver has converted all of the input clauses into constraints, it passes control to the PB solver. From that point on, the SAT solver serves in a support role, generating proofs to justify the steps of the PB solver. As the PB solver operates, it generates a BDD representation of each new constraint: for each equation $\mathbf{e}'_i$ generated by Gaussian elimination (1) or each ordering constraint $\mathbf{c}_{i,i'}$ generated by Fourier-Motzkin elimination (2). For a new BDD with root u_{n+1} generated from constraints represented by terms T_i and T_j, it uses the APPLYAND operation to generate the conjunction w of the BDDs with roots u_i and u_j, as well as a proof of the clause $\overline{u}_i \vee \overline{u}_j \vee w$. It then uses the PROVEIMPLICATION operation with arguments w and u_{n+1} to generate a proof of the clause $\overline{w} \vee u_{n+1}$. It can then resolve the unit clauses for terms T_i and T_j with the generated clauses to generate a proof of the unit clause u_{n+1}, and so the BDD representation of the constraint becomes term T_{n+1}. When some step of the PB solver generates an unsolvable equation or ordering constraint, it encodes the constraint as the *false* BDD leaf $\bot$, and the SAT solver will generate the empty clause.

As an optimization, we implemented an operation APPLYANDPROVEIMPLICATION combining the functions of APPLYAND and PROVEIMPLICATION. It takes as arguments BDD roots u, v, and w and generates a proof that $u \wedge v \rightarrow w$ without constructing the BDD representation of $u \wedge v$. We found this reduced the total proof lengths by over 2×.

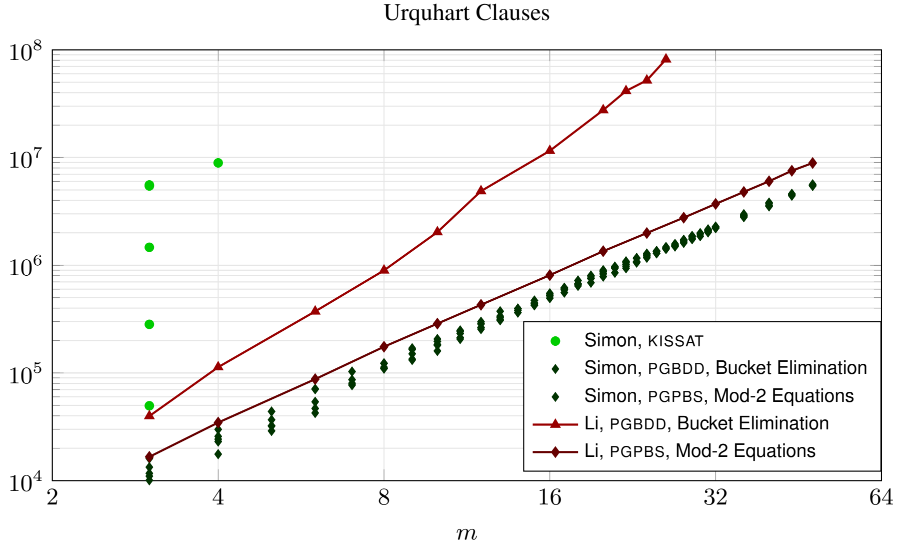


Fig. 4. Total number of clauses in proofs of two sets of Urquhart formulas.

4 Experimental Results

PGPBS is written in Python with its own BDD package and pseudo-Boolean constraint solver.[3] The Gaussian elimination solver employs a standard greedy pivot selection heuristic, attributed to Markowitz [23, 41], that seeks to minimize the number of nonzero coefficients created. The Fourier-Motzin solver uses a similar heuristic for selecting pivot variables.

The operation of PGPBS follows the flow illustrated in Figure 2, with constraints extracted directly from the input CNF file, and with the generated schedule driving the operation of the solver. Some measurements were taken using a BDD variable ordering according to their numbering in the input file, while others used a random BDD variable ordering to assess the sensitivity to the variable ordering. All generated proofs were checked with an LRAT proof checker [20]. We used KISSAT, winner of the 2020 SAT competition [7], as a representative CDCL solver. All measurements labeled "PGBDD" are for the earlier version of the solver, without pseudo-Boolean reasoning [11, 12].

We measure the performance of the solvers in terms of the total number of clauses in the generated proofs of unsatisfiability. This metric tracks closely with the solver runtime and has the advantage that it is machine independent. We set an upper limit of 100 million clauses for the proof sizes for the three measured solvers.

4.1 Urquhart Parity Formulas

Urquhart [49] defined a family of formulas that require resolution proofs of exponential size. Over the years, two sets of SAT benchmarks have been labeled as "Urquhart Prob-

[3] PGPBS, PGBDD, and the code for generating and testing a set of benchmarks, are available at https://github.com/rebryant/pgpbs-artifact and as https://doi.org/10.5281/zenodo.5907086.

lems" [15, 38]. The formulas are defined over a class of degree-5, undirected, bipartite graphs, parameterized by a size m, with the graph having $2m^2$ nodes. To transform a graph into a formula, each edge $\{i, j\}$ in the set of edges E has an associated variable $x_{\{i,j\}}$. (We use set notation to emphasize that the order of the indices does not matter.) Each vertex is assigned a polarity $p_i \in \{0, 1\}$, such that the sum of the polarities is odd. The clauses then encode that the sum for all values of i and j of $x_{\{i,j\}} + p_i$ equals 0 modulo 2. This is false of course, since each edge is counted twice in the sum, and the sum of the polarities is odd.

The two families of benchmarks differ in how the graphs are constructed. Li's benchmarks are based on the explicit construction of *expander* graphs [26, 40], upon which Urquhart's lower bound proof is based. Simon's benchmarks are based on randomly generated graphs and thus depend on the random seed. We generated five different formulas for each value of m. Simon's graphs are not guaranteed satisfy the expander property, but they still provide challenging benchmarks for SAT solvers.

Figure 4 shows the performance of the solvers, measured as the number of clauses as a function of m, for both Simon's and Li's benchmarks. The smallest instances of the benchmark have $m = 3$. As can be seen KISSAT is able to generate proofs for the Simon version for four cases with $m = 3$ and one with $m = 4$, but it is unable to handle any other cases, including not even the minimum instance for Li's benchmark. Measurements are shown for PGBDD running bucket elimination, a simple algorithm that processes clauses and intermediate terms with conjunction and quantification operations according to the levels of the topmost variables [22, 35]. It achieves polynomial scaling on both benchmarks, with only mild sensitivity to the random seeds. Running PGPBS with modulo-2 equation solving improves the performance even further, such that we were able to handle both families of benchmarks up to $m = 48$. Considering that the problem grows quadratically in m, this represents a major improvement over KISSAT.

4.2 Other Parity Constraint Benchmarks

Chew and Heule [17] introduced a benchmark based on Boolean expressions computing the parity of a set of Boolean values $x_1, \ldots, x_n$ using two different orderings of the inputs, with a randomly chosen variable negated in the second computation. The SAT problem is to find a satisfying assignment that makes the two expressions yield the same result—an impossibility due to the negated variable. With KISSAT, we found the results were very sensitive to the choice of random permutation, and so we ran the solver for five different random seeds for each value of n. We were able to generate proofs for instances with n up to 47, but we also encountered cases where the proofs exceeded the 100-million clause limit starting with $n = 40$. The overall scaling is exponential.

Chew and Heule showed they could generate proofs for this problem that scale as $n \log n$. Using bucket elimination, PGBDD is able to obtain polynomial performance, handling up to $n = 3{,}000$ with a proof of 61 million clauses. PGPBS is able to apply Gaussian elimination with modulus $r = 2$, obtaining even better performance than did Chew and Heule. For $n = 10{,}000$, Chew and Heule's proof has 14 million clauses while the proof generated by PGPBS has less than 7 million.

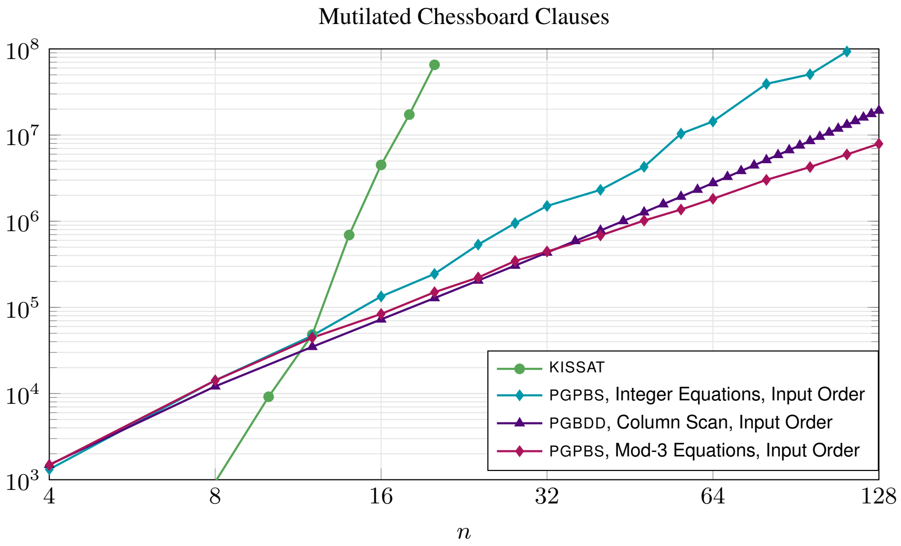


Fig. 5. Total number of clauses in proofs of $n \times n$ mutilated chess board problems.

Elffers and Nordström created the TSEITINGRID family of benchmarks for the 2016 SAT competition, based on grid graphs having fixed width but variable lengths [25]. These are designed to be challenging for SAT solvers while having polynomial scaling. The 2020 SAT competition included two instances of this benchmark, with 7×165 and 7×185 grids. None of the entrants could generate an unsatisfiability proof for either instance within the 5000 second time limit. On the other hand, PGPBS can readily handle both, generating proofs with less than 500,000 clauses and requiring at most 63 seconds. Indeed, PGPBS can solve the largest published instance, having a 7×200 grid, in 76 seconds. Clearly, parity constraint problems pose no major challenge for PGPBS.

4.3 Variants of the Mutilated Chessboard

The mutilated chessboard problem considers an $n \times n$ chessboard, with the corners on the upper left and the lower right removed. It attempts to tile the board with dominos, with each domino covering two squares. Since the two removed squares had the same color, and each domino covers one white and one black square, no tiling is possible. This problem has been well studied in the context of resolution proofs, for which it can be shown that any proof must be of exponential size [2].

The standard CNF encoding defines a Boolean variable for each possible horizontal or vertical domino placement. For each square, it encodes an exactly-one constraint for the set of dominos that could cover that square. Both the number of variables and the number of clauses scale as $\Theta(n^2)$. Figure 5 shows the performance of the different solvers as a function of n. KISSAT scales exponentially, hitting the 100-million clause limit with $n = 20$. The plot labeled "Column Scan" demonstrates that PGBDD performs very well on this problem when given a carefully crafted schedule and the proper variable ordering [11], requiring less than 20 million clauses for $n = 128$.

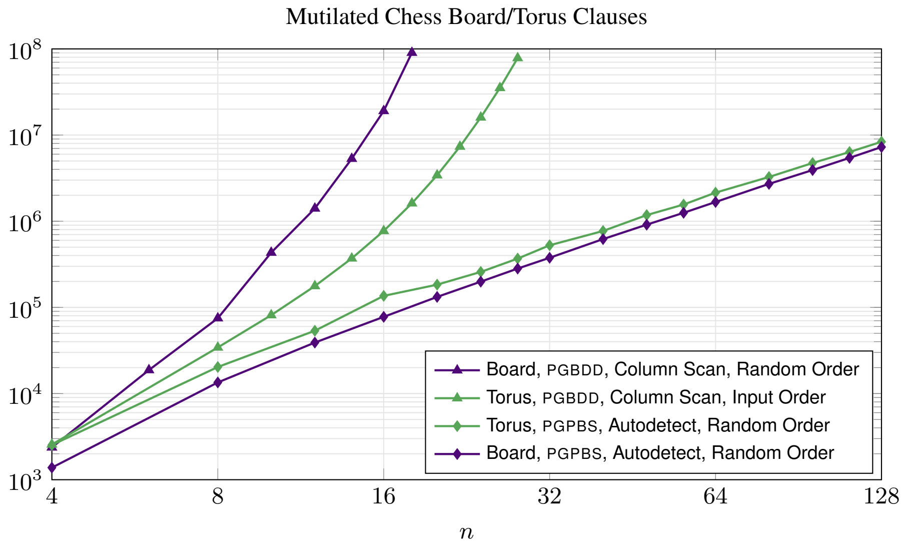


Fig. 6. Stress Testing: Changing the topology and variable ordering for mutilated chess. Autodetection enables the PB solver to use modulo-3 arithmetic.

The plot labeled "Integer Equations, Input Ordering" shows that PGPBS can achieve polynomial scaling on this problem when performing Gaussian elimination on integer equations. It does not scale as well as column scanning, reaching $n = 96$ before hitting the clause limit. (The unevenness of the plot appears to be an artifact of the randomization used to break ties during pivot selection.)

Looking deeper, we can see that solver avoids the worst-case performance for Gaussian elimination on this problem. Let us assume that the omitted corners are both white, and so the board has k black squares and $k - 2$ white squares, where $k = n^2/2$. Each variable occurs in one equation for a black square and in one for a white square. If we were to sum all of the equations for the black squares, we would get $\sum_{j=1,m} x_j = k$, where m is the number of variables. Similarly, summing the equations for the white squares gives $\sum_{j=1,m} x_j = k - 2$. Subtracting the second equation for the first gives the unsolvable equation $0 = 2$. These sums and differences can be performed using pseudo-Boolean equations with coefficients 0 and $+1$. Although Gaussian elimination combines equations in a different order, it maintains the property that the coefficients are limited to values -1, 0, and $+1$.

The plot labeled "Mod-3 Equations, Input Ordering" demonstrates the benefit of modular arithmetic when solving systems of equations. The equation $0 = 2$, obtained by integer Gaussian elimination for this problem, has no solution for any odd modulus; modulus auto-detection chooses $r = 3$. This optimization achieves better scaling, due to the bounded width of the BDD representations. Indeed, it outperforms the best results obtained with PGBDD, generating a proof with less than 8 million clauses for $n = 128$. For the remaining measurements, we assume that modulus auto-detection is enabled.

The plots of Figure 6 illustrate how pseudo-Boolean reasoning makes PGPBS more robust than PGBDD. First, we consider the extension of the mutilated chessboard prob-

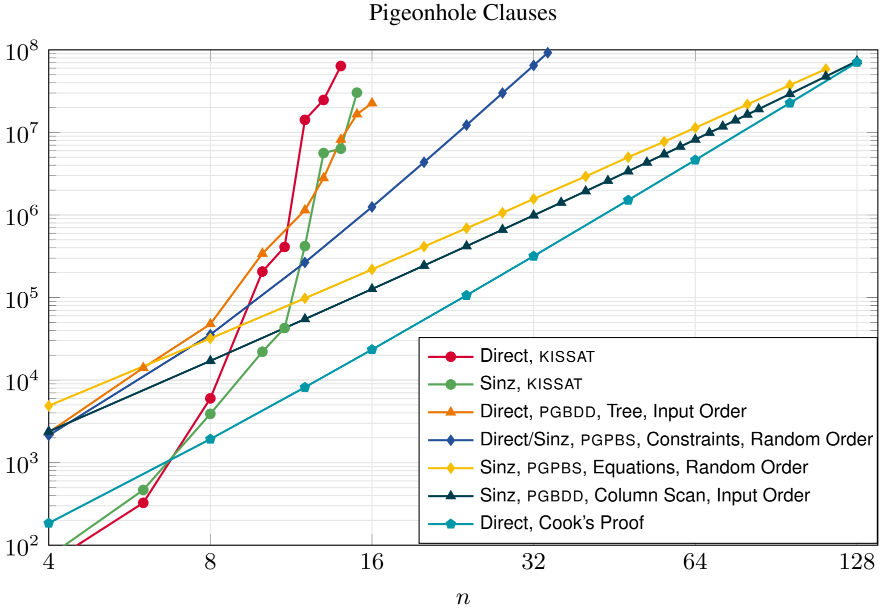


Fig. 7. Total number of clauses in proofs of pigeonhole problem for n holes

lem to a *torus*, with the sides of the board wrapping around both vertically and horizontally. As the plot labeled "Torus, PGBDD, Column Scan, Input Order" indicates, the performance of column scanning disintegrates for this seemingly minor change. The compact state encoding exploited by column scanning works only when there is a single frontier as the variables are processed from left to right. Second, the plot labeled "Board, PGBDD, Column Scan, Random Order" illustrates that column scanning is highly sensitive to the chosen BDD variable ordering. On the other hand, the four versions using auto-detected modular equations are only mildly sensitive to the topology (torus or board) or the variable ordering (input or random). For both topologies, the clause counts for the two different orderings (input and random) are so close to each other that they cannot be distinguished on the log-log scale. and so we show only the results for random orderings. These results show that pseudo-Boolean reasoning overcomes several major weaknesses of the pure Boolean methods of PGBDD. With its PB solver, PGPBS requires no guidance from the user regarding how to process the clauses, nor does it require any guidance or heuristics to choose a good BDD variable ordering. Furthermore, it is less sensitive to the problem definition.

4.4 Pigeonhole Problem

The pigeonhole problem is one of the most studied problems in propositional reasoning. Given a set of n holes and a set of $n+1$ pigeons, it asks whether there is an assignment of pigeons to holes such that (1) every pigeon is in some hole, and (2) every hole contains at most one pigeon. The answer is no, of course, but any resolution proof for this must be of exponential length [29].

The problem can be encoded into CNF with Boolean variables $p_{i,j}$, for $1 \leq i \leq n$ and $1 \leq j \leq n+1$, indicating that pigeon j is placed in hole i. A set of n AMO constraints indicates that each hole can contain at most one pigeon, and $n+1$ ALO constraints indicate that each pigeon must be placed in some hole. We experimented with two different encodings for the AMO constraints: the direct encoding requiring $n\,(n+1)/2$ clauses per hole, and the Sinz encoding [45], requiring $3n-1$ clauses.

Figure 7 shows the total number of clauses (input plus proof) as functions of n for this problem. KISSAT performs poorly, reaching the 100-million clause limit with $n = 14$ for the direct encoding and $n = 15$ for the Sinz encoding. Using PGBDD, we were unable to find any strategy that gets beyond $n = 16$ with a direct encoding. Our best results came from a "tree" strategy, simply forming the conjunction of the input clauses using a balanced tree of binary operations. For the Sinz encoding, on the other hand, we devised a column scanning technique similar to the method used to solve the mutilated chessboard problem. This approach scales very well, empirically measured as $\Theta(n^3)$. The proofs stay below 100 million clauses up to $n = 128$, although it can only reach $n = 17$ with a random variable ordering (plot not shown).

Using pseudo-Boolean reasoning with Fourier-Motzkin elimination, we were able to achieve polynomial scaling, reaching $n = 34$ with both encodings and for both input and random ordering. The four results are so similar that they are indistinguishable on a log-log plot, and so we show the average for the two encodings with random orderings. Observe that each variable $p_{i,j}$ occurs with coefficient -1 in the AMO constraint for hole i and with coefficient $+1$ in the ALO constraint for pigeon j. Thus, as described in Section 2.3, each step of Fourier-Motzkin elimination reduces the number of constraints by at least one, with the coefficients restricted to the values -1, 0, and $+1$. Indeed, it can be seen that the solver, in effect, sums the n AMO and $n+1$ ALO constraints to get the unsolvable constraint $0 \geq 1$. The scaling of proof sizes, empirically measured as $\Theta(n^5)$, is limited by the $O(n^2)$ growth of the BDD representations for the ordering constraints, as was illustrated in Figure 1C.

The plot labeled "Sinz, PGPBS, Equations, Random Order" demonstrates the effect of adding constraints to enforce exactly-one constraints on both the pigeons and the holes. The solver applies modulus auto-detection to give a modulus of $r = 2$. Modulo-2 reasoning enables the solver to match the performance of column scanning, with the further advantages of being fully automated and being insensitive to the variable ordering. However, it requires additional constraints in the input file.

Finally, the plot labeled "Direct, Cook's Proof" shows the complexity of Cook's extended-resolution proof of the pigeonhole problem [19], encoded in DRAT format. Although it is very concise for small values of n, its scaling as $\Theta(n^4)$ lies between the $\Theta(n^3)$ achieved by column scanning and equation solving, and the $\Theta(n^5)$ achieved by constraint solving. Of these, only Cook's proof and the solution by constraint solving are directly comparable, in that only these use a direct encoding and have only the minimum set of AMO and ALO constraints.

In summary, pseudo-Boolean reasoning makes this problem tractable with full automation, and it has minimal sensitivity to the variable ordering. Generating proofs by solving systems of ordering constraints is more challenging than by solving automatically detected modular equations, but both achieve polynomial scaling.

4.5 Other Cardinality Constraint Problems

Codel et al. [18] defined a general class of problems that includes the mutilated chessboard and the pigeonhole problems as special cases. Given a bipartite graph with vertices L and R such that $|L| < |R|$, the problem is to find a perfect matching, i.e., a subset of the edges such that each vertex has exactly one incident edge. For the mutilated chessboard, L and R correspond to the white and black squares, respectively, with edges based on chessboard adjacencies. For pigeonhole, L corresponds to the holes and R to the pigeons, and the graph is the complete bipartite graph $K_{n,n+1}$. No instance of this matching problem has a solution, since the sets of nodes are of unequal size.

Twelve instances of this problem were included in the 2021 SAT competition, based on randomly generated graphs with $n = |L|$ ranging from 15 to 20 and with $|R| = n+1$. Different methods were used to encode the AMO constraints, and some included clauses to convert both sets of constraints into exactly-one constraints. In the competition, all of the solvers could easily handle the benchmarks with $n = 15$, most could handle $n = 16$, with typical runtimes of around 1000 seconds, but none could solve any of the larger problems. PGPBS can easily handle all of the benchmarks, requiring at most 13 seconds and generating proofs with less than 500,000 clauses.

5 Conclusions

Incorporating pseudo-Boolean reasoning into a SAT solver enables it to handle classes of problems encoded in CNF that are intractable for CDCL solvers. By having the PB solver generate BDD representations of its intermediate results, a BDD-based, proof-generating SAT solver can generate clausal proofs of unsatisfiability on behalf of the PB solver in the standard, DRAT proof framework. Compared to the SAT solver operating on its own, including a PB solver enables greater automation with less sensitivity to problem definition, encoding method, and variable ordering.

We have shown that applying pseudo-Boolean reasoning to unsatisfiable instances of parity and cardinality constraint problems can yield proofs that scale polynomially. Solving systems of equations over the integers modulo 2 yields 0-1 valued solutions, and so parity reasoning can also be used on satisfiable problems [6, 30, 37, 47]. On the other hand, Gaussian elimination over integers or with modulus $r > 2$, as well as Fourier-Motzkin elimination, are not guaranteed to find 0-1 valued solutions. When seeking solutions with cardinality reasoning, it seems more effective to use methods that adapt CDCL-based search to pseudo-Boolean constraints [14].

The method described here can be generalized to incorporate other reasoning methods into a proof-generating SAT solver. As long as intermediate results can be expressed as BDDs, a proof can be generated that the result of each step logically follows from the preceding steps. Thus, we could incorporate other pseudo-Boolean reasoning methods, such as cutting planes [28, 32], or we could add totally different reasoning methods.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A new look at BDDs for pseudo-Boolean constraints. Journal of Artificial Intelligence Research **45**, 443–480 (2012)
2. Alekhnovich, M.: Mutilated chessboard problem is exponentially hard for resolution. Theoretical Computer Science **310**(1-3), 513–525 (Jan 2004)
3. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: AAAI Conference on Artificial Intelligence. pp. 15–20 (2010)
4. Bareiss, E.H.: Sylvester's identity and multistep integer-preserving Gaussian elimination. Mathematics of Computation **22**, 565–578 (1968)
5. Barnett, L.A., Biere, A.: Non-clausal redundancy properties. In: Conference on Automated Deduction (CADE). LNAI, vol. 12699, pp. 252–272 (2021)
6. Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In: Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions. Dep. of Computer Science Series of Publications B, vol. B-2016-1, pp. 44–45. University of Helsinki (2016)
7. 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)
8. Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing SAT solving. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 391–435. IOS Press, second edn. (2021)
9. Biere, A., Le Berre, D., Lonca, E., Manthey, N.: Detecting cardinality constraints in CNF. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 285–301 (2014)
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)
11. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I. LNCS, vol. 12651, pp. 76–93 (2021)
12. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. CoRR **abs/2105.00885** (2021)
13. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transition relations. In: VLSI91 (1991)
14. Chai, D., Kuehlmann, A.: A fast pseudo-Boolean constraint solver. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**(3), 305–317 (2005)
15. Chatalic, P., Simon, L.: ZRes: The old Davis-Putnam procedure meets ZBDD. In: Conference on Automated Deduction (CADE). LNCS, vol. 1831, pp. 449–454 (2000)
16. Chen, J.: A new SAT encoding of at-most-one constraint. In: Workshop on Constraint Modeling and Reformulation (2010)
17. Chew, L., Heule, M.J.H.: Sorting parity encodings by reusing variables. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 12178, pp. 1–10 (2020)
18. Codel, C., Reeves, J., Heule, M.J.H., Bryant, R.E.: Bipartite perfect matching benchmarks. In: Pragmatics of SAT (2021)
19. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News **8**(4), 28–32 (Oct 1976)
20. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: Conference on Automated Deduction (CADE). LNCS, vol. 10395, pp. 220–236 (2017)

21. Dantzig, G.B., Eaves, B.C.: Fourier-Motzkin elimination and its dual with application to integer programming. In: Combinatorial Programming: Methods and Applications. pp. 93–102. Springer (1974)
22. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1–2), 41–85 (1999)
23. Duff, I.S., Reid, J.K.: A comparison of sparsity orderings for obtaining a pivotal sequence in Gaussian elimination. IMA Journal of Applied Mathematics **14**(3), 281–291 (1974)
24. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal of Satisfiability, Boolean Modeling and Computation **2**, 1–26 (2006)
25. Ellfers, J., Nordström, J.: Documentation of some combinatorial benchmarks. In: Proceedings of the SAT Competition 2016 (2016)
26. Gabber, O., Galil, Z.: Explicit construction of linear-sized superconcentrators. Journal of Computer and System Sciences **22**, 407–420 (1981)
27. Gocht, S., Nordström, J.: Certifying parity reasoning efficiently using pseudo-Boolean proofs. In: AAAI Conference on Artificial Intelligence. pp. 3768–3777 (2021)
28. Gomory, R.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64**, 275–278 (1958)
29. Haken, A.: The intractability of resolution. Theoretical Computer Science **39**, 297–308 (1985)
30. Han, C.S., Jiang, J.H.R.: When Boolean satisfiability meets Gaussian elimination in a simplex way. In: Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 410–426 (2012)
31. Heule, M.J.H., Hunt, Jr., W.A., Wetzler, N.D.: Verifying refutations with extended resolution. In: Conference on Automated Deduction (CADE). LNCS, vol. 7898, pp. 345–359 (2013)
32. Hooker, J.N.: Generalized resolution and cutting planes. Annals of Operations Research **12**, 217–238 (1988)
33. Hosaka, K., Takenaga, Y., Yajima, S.: Size of ordered binary decision diagrams representing threshold functions. Theoretical Computer Science **180**, 47–60 (1996)
34. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370 (2012)
35. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 54–60 (2006)
36. Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from N objects. In: Constraints in Formal Verification (CFV) (2007)
37. Laitinen, T., Junttila, T., Niemelä, I.: Extending clause learning SAT solvers with complete parity reasoning. In: International Conference on Tools with Artificial Intelligence. pp. 65–72. IEEE (2012)
38. Li, C.M.: Equivalent literal propagation in the DLL procedure. Discrete Applied Mathematics **130**(2), 251–276 (2003)
39. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of Boolean formulas. In: Haifa Verification Conference. LNCS, vol. 7857 (2013)
40. Margulis, G.A.: Explicit construction of concentrators. Probl. Perdachi Info (Problems in Information Transmission) **9**(4), 71–80 (1973)
41. Markowitz, H.M.: The elimination form of the inverse and its application to linear programming. Management Science **3**(3), 213–284 (1957)
42. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
43. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J.ACM **12**(1), 23–41 (January 1965)
44. Rosser, J.B.: A method of computing exact inverses of matrices with integer coefficients. Journal of Research of the National Bureau of Standards **49**(5), 349–358 (1952)

45. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: Principles and Practice of Constraint Programming (CP). LNCS, vol. 3709, pp. 827–831 (2005)
46. Sinz, C., Biere, A.: Extended resolution proofs for conjoining BDDs. In: Computer Science Symposium in Russia (CSR). LNCS, vol. 3967, pp. 600–611 (2006)
47. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Proc. of the 12th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2009). LNCS, vol. 5584, pp. 244–257 (2009)
48. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970. pp. 466–483. Springer (1983)
49. Urquhart, A.: The complexity of propositional proofs. The Bulletin of Symbolic Logic **1**(4), 425–467 (1995)
50. Wetzler, N.D., Heule, M.J.H., Hunt Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 422–429 (2014)
51. Williams, H.P.: Fourier-Motzkin elimination extension to integer programming problems. Journal of Combinatorial Theory (A) **21**, 118–123 (1976)
52. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: Design, Automation and Test in Europe (DATE). pp. 880–885 (2003)

Moving Definition Variables in Quantified Boolean Formulas[★]

Joseph E. Reeves ✉, Marijn J. H. Heule, and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
{jereeves,mheule,randy.bryant}@cs.cmu.edu

Abstract. Augmenting problem variables in a quantified Boolean formula with definition variables enables a compact representation in clausal form. Generally these definition variables are placed in the innermost quantifier level. To restore some structural information, we introduce a preprocessing technique that moves definition variables to the quantifier level closest to the variables that define them. We express the movement in the QRAT proof system to allow verification by independent proof checkers. We evaluated definition variable movement on the QBFEVAL'20 competition benchmarks. Movement significantly improved performance for the competition's top solvers. Combining variable movement with the preprocessor BLOQQER improves solver performance compared to using BLOQQER alone.

1 Introduction

Boolean formulas and circuits can be translated into conjunctive normal form (CNF) by introducing *definition variables* to augment the existing *problem variables*. Definition variables are introduced through a set of *defining clauses*, given by the Tseitin [19] or Plaisted-Greenbaum [16] transformation. Problem variables occurring in the defining clauses constitute the *defining variables*; they effectively determine the values of the definition variables. In CNF, definitions are not an explicit part of the problem representation, preventing solvers from using this structural information. Quantified Boolean formulas (QBF) extend CNF into prenex conjunctive normal form (PCNF) with the addition of quantifier levels. In practice, definition variables are usually placed in the innermost quantifier level. However, as we will show, placing a definition variable in the quantifier level immediately following its defining variables can improve solver performance.

We describe a preprocessing technique for moving definition variables to the quantifier level of their innermost defining variables. As a starting point, existing tools KISSAT and CNFTOOLS can detect definitions in a CNF formula. We process and order the candidate definitions, moving definition variables sequentially. For each instance of movement we generate a proof in the QRAT proof system that, through a series of clause additions and deletions, effectively replaces the old definition variable with a new variable at the desired quantification level.

[★] The authors are supported by the NSF under grant CCF-2108521.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 462–479, 2022.
https://doi.org/10.1007/978-3-030-99524-9_26

Most Boolean satisfiability (SAT) solvers generate proofs of unsatisfiability for independent checking [7,9,20]. This has proved valuable for verifying solutions independent of the (potentially buggy) solvers. Proof generation is difficult for QBF and relatively uncommon in solvers. The QBF preprocessor BLOQQER [2] generates QRAT proofs [8] for all of the transformations it performs. Our QRAT proofs for variable movement also allow verification with the independent proof checker QRAT-TRIM, ensuring that the movement preserves equivalence with the original formula.

Clausal-based QBF solvers rely on preprocessing to improve performance. Almost every top-tier solver in the QBFEVAL'20 competition[1] used some combination of BLOQQER, HQSPRE [21], or QBFRELAY [15]. Some solvers incorporate preprocessing techniques into the solving phase, e.g., DEPQBF's [14] use of dynamic quantified blocked clause elimination. Unlike other preprocessing techniques, variable movement does not add or remove clauses or literals. However, it can prompt the removal of literals through universal reduction and may guide solver decisions in a beneficial way.

The contributions of this paper include: (1) adapting the SAT solver KISSAT and CNF preprocessor CNFTOOLS to detect definitions in a QBF, (2) giving an algorithm for moving variables that maximizes variable movement, (3) formulating steps for generating a QRAT proof of variable movement, and (4) evaluting the impact of these transformations. Variable movement significantly improves the performance of top solvers from the QBFEVAL'20 competition. Combining variable movement with BLOQQER further improves solver performance.

2 Preliminaries

2.1 Quantified Boolean Formulas

Quantified Boolean formulas (QBF) can be represented in prenex conjunctive normal form (PCNF) as $\Pi.\psi$, where Π is a prefix of the form $Q_1 X_1 Q_2 X_2 \cdots Q_n X_n$ for $Q_i \in \{\forall, \exists\}$ and the matrix ψ is a CNF formula. The formula ψ is a conjunction of clauses, where each clause is a disjunction of literals. A literal l is either a variable $l = x$ or negated variable $l = \overline{x}$, and $Var(l) = x$. The formula $\psi(l)$ is the clauses $\{C \mid C \in \psi, l \in C\}$. The set of all variables occurring in a formula is given by $Var(\psi)$. Substituting a variable y for x in ψ, denoted as $\psi[y/x]$, will replace every instance of x with y and $\overline{x}$ with $\overline{y}$ in the formula. The sets of variables X_i are disjoint, and we assume every variable occurring in ψ is in some X_i. A variable x is *fresh* if it does not occur in $\Pi.\psi$. The quantifier for literal l with $Var(l) \in X_i$ is $\mathcal{Q}(\Pi, l) = Q_i$, and l is said to be in *quantifier level* $\lambda(l) = i$. If $\mathcal{Q}(\Pi, l) = Q_i$ and $\mathcal{Q}(\Pi, k) = Q_j$, then $l \leq_\Pi k$ if $i \leq j$. $Q_1 X_1$ is referred to as the outermost quantifier level and $Q_n X_n$ is the innermost quantifier level.

2.2 Inference Techniques in QBF

Given a clause C, if a literal $l \in C$ is universally quantified, and all existentially quantified literals $k \in C$ satisfy $k <_\Pi l$, then l can be removed from C. This process is

[1] available at http://www.qbflib.org/qbfeval20.php

called *universal reduction* (UR). Given two clauses C and D with $x \in C$ and $\overline{x} \in D$, the *Q-resolvent* over *pivot* variable x is $\mathrm{UR}(C) \cup \mathrm{UR}(D) \setminus \{x, \overline{x}\}$ [12]. The operation is undefined if the result is tautological. This extends *resolution* for propositional logic by applying UR to the clauses before combining them, while disallowing tautologies. Adding or removing non-tautological Q-resolvents preserves logical equivalence.

Given a prefix Π and clauses C and D with $l \in C$ and $\bar{l} \in D$, the *outer resolvent* over existentially quantified *pivot* literal l is $C \cup \{k \mid k \in D, k \neq \bar{l}, k \leq_{\Pi} l\}$. Given a QBF $\Pi.\psi$, a clause C is *Q-blocked* on some existentially quantified literal $l \in C$ if for all $D \in \psi(\bar{l})$ the outer resolvent of C with D on l is a tautology. This extends the *blocked* property for CNF with the restriction on the conflicting literal's quantifier level.

A clause C *subsumes* D if $C \subseteq D$. The property *Q-blocked-subsumed* generalizes Q-blocked by requiring the outer resolvents be tautologies or subsumed by some clause in the formula.

Given a QBF $\Psi = \Pi.\psi$, if a clause C is Q-blocked-subsumed then C is QRAT w.r.t. Ψ. In this case, C can be added to ψ or if $C \in \psi$ deleted from ψ while preserving equivalence. A series of clause additions and deletions resulting in the empty formula is a *satisfaction proof* for a QBF if all clause deletions are QRAT. A series of clause additions and deletions deriving the empty clause is a *refutation proof* for a QBF if all clause additions are QRAT. If both clause additions and deletions are QRAT, each step preserves equivalence regardless of the truth value of the QBF. We call this a *dual proof*. The QBF Ψ' that results from applying the dual proof steps to Ψ is equivalent to Ψ.

2.3 Definitions

A variable x is a definition variable in $\Psi = \Pi.\psi$ with defining clauses $\delta(x)$ containing x, $\delta(\bar{x})$ containing $\overline{x}$, and defining variables $Z_x = \mathit{Var}[\delta(x) \cup \delta(\overline{x})] \setminus \{x\}$ when two properties hold: (1) the definition is *left-total*, meaning that for every assignment of Z_x there exists a value of x that satisfies $\delta(x) \cup \delta(\bar{x})$, and (2) the definition is *right-unique*, meaning that for every assignment of Z_x there exists exactly one value of x that satisfies $\delta(x) \cup \delta(\bar{x})$. The clauses $\delta(x) \cup \delta(\bar{x})$ are left-total iff they are Q-blocked on variable x. This implies that the definition variable comes after the defining variables w.r.t. Π. The definition is right-unique if the SAT problem $\{C \setminus \{x, \overline{x}\} \mid C \in \delta(x) \cup \delta(\bar{x})\}$ is unsatisfiable. We can assume that any right-unique variable is existentially quantified, otherwise the formula would be trivially false.

The *remaining clauses* of x are $\rho(x) = \psi(x) \setminus \delta(x)$ and $\rho(\overline{x}) = \psi(\overline{x}) \setminus \delta(\bar{x})$. If x occurs as a single polarity in the remaining clauses, it can be encoded as a *one-sided* definition: if $\rho(x)$ is empty only $\delta(x)$ are needed to determine if x is assigned to true and therefore unable to satisfy the clauses in $\rho(\overline{x})$. This is a stronger condition than *monotonicity* used for the general Plaisted-Greenbaum transformation [16].

Example 1. $x \leftrightarrow a \wedge b$ is written in CNF as $(x \vee a \vee b) \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee b)$. Given $\rho(x) = \{(x \vee c), (x \vee d \vee e)\}$ and $\rho(\overline{x}) = \{\}$, $x = a \wedge b$ can be written as a one-sided definition with clauses $(\overline{x} \vee a) \wedge (\overline{x} \vee b)$.

In some definitions including exclusive-or (XOR denoted by $\oplus$), multiple variables are left-total and right-unique. Determining the definition variable requires information about how definition variables are nested within the formula.

Q-resolution can be generalized to sets of clauses $\mathcal{C}$ and $\mathcal{D}$, denoted $\mathcal{C} \otimes_x \mathcal{D}$, by generating the non-tautological resolvents from clauses in $\mathcal{C}(x)$ and $\mathcal{D}(\overline{x})$ on pivot variable x pairwise. Given a definition variable x and defining variables $\{z_1, \ldots, z_n\}$, let x' be a fresh variable with $\theta_x = \delta(x)$ and $\theta_{\overline{x}'} = \delta(\overline{x})[x'/x]$. The procedure *defining variable elimination* applies set-based Q-resolution in the following way: set $\theta_1 = \theta_x(z_1) \otimes_{z_1} \theta_{\overline{x}'}(\overline{z}_1) \wedge \theta_x(\overline{z}_1) \otimes_{z_1} \theta_{\overline{x}'}(z_1)$ and compute $\theta_2 = \theta_1(z_2) \otimes_{z_2} \theta_1(\overline{z}_2)$; continue the process until $\theta_n = \theta_{n-1}(z_n) \otimes_{z_n} \theta_{n-1}(\overline{z}_n)$. UR is not applied because x is in the innermost quantifier level with respect to its defining variables. The first step ensures all clauses in θ_1 will contain both x and x'. θ_n will either be $\{(x \vee x')\}$ or empty. If $\theta_n = \{(x \vee x')\}$, linearizing the sets of resolvents θ_i forms a Q-resolution derivation of $(x \vee x')$. This is similar to Davis Putnam variable elimination [4].

3 Definition Detection

Given a QBF with no additional information, we first detect definitions to determine which variables can be moved. All definitions are detected before variable movement begins. Variable movement depends on the defining clauses, the definition variables, and the nesting of definition variables. At a minimum, definition detection must produce the defining clauses, and the rest can be inferred during movement.

Since the seminal work by Eén and Biere [5], bounded variable elimination (BVE) has been an essential preprocessing technique in SAT solving. The technique relies on definitions, so most SAT solvers incorporate some form of definition detection. The conflict-driven clause learning SAT solver KISSAT [1] extends the commonly used syntactic pattern matching with semantic definition detection. The detection is applied to variables independently. Alternatively, the preprocessor CNFTOOLS [10] performs hierarchical definition detection, capturing additional information about definition variable nesting and monotonic definitions.

These tools run on CNF formulas. A QBF can be transformed into a CNF by removing the prefix, but not all definitions in the CNF are valid w.r.t. the prefix. For example, some definitions will not be left-total because of the quantifier level restrictions in the Q-blocked property. Such definitions can be easily filtered out before variable movement, so there is no need to add these quantifier-based checks into the tools.

3.1 Hierarchical Definition Detection in CNFTOOLS

The hierarchical definition detection in CNFTOOLS employs a breadth first search (BFS) to recurse through nested definitions in a formula. Root clauses are selected heuristically, then BFS begins on the variables occurring in those clauses. All unit clauses are selected as root clauses. The *max-var* heuristic selects root variables based on their numbering. This exploits the practice of numbering definition variables after problem variables. The more involved *min-unblocked* heuristic finds a minimally unblocked literal. This is more expensive to compute but does not rely on variable numbering.

When a variable is encountered in the BFS, CNFTOOLS checks if the defining clauses are blocked. If so, the following detection methods are applied: pattern matching for BiEQ, AND, OR, and full patterns, monotonic checking, and semantic checking. BiEQ refers to an equivalence between two variables.

A definition is a full pattern if $\forall C \in \delta(x) \cup \delta(\overline{x})$, $|C| = n + 1$ where n is the number of defining variables and there are 2^n defining clauses. The full pattern includes some common encodings for XOR, XNOR, NOT, and Majority3, but is often avoided. Since the detection follows the hierarchical nesting of definitions, there is no ambiguity between the defining variables and definition variables in XOR definitions.

The advantage of hierarchical detection is the ability to detect monotonic definitions. For variable movement we consider only monotonic definitions that are either fully-defined or one-sided. If a monotonic definition is not fully-defined but the definition variable occurs positively and negatively in the defining clauses of other definitions, the additional clauses can prevent variable movement w.r.t. the QRAT proof system.

Semantic checking involves solving the SAT problem for right uniquness described in the preliminaries. As definitions are detected the defining clauses are removed from the formula for the following iterations. This can produce problematic one-sided definitions. For example, a variable may occur both positively and negatively in the defining clauses of other definitions, and removing those clauses makes the variable one-sided. Similar to the monotonic case, the additional defining clauses can prevent movement w.r.t. the QRAT proof system, so these types of definitions must be filtered out.

3.2 Independent Definition Detection in KISSAT

KISSAT uses definition detection to find candidates for BVE. Starting with the 2021 SAT Competition, KISSAT added semantic definition detection [6] to complement the existing syntactic pattern matching for BiEQ, AND, OR, ITE, and XOR definitions. In semantic detection an internal SAT solver KITTEN with low overhead and limited capabilities performs a right-uniqueness check on the formula $\psi(x) \cup \psi(\overline{x})$ after removing all occurrences of x and $\overline{x}$. This formula includes $\rho(x)$ and $\rho(\bar{x})$ as the set of defining clauses are not known in advance. If the formula is unsatisfiable, an unsatisfiable core is extracted (potentially after reduction) and returned as the set of defining clauses.

Core extraction does not guarantee the defining clauses are blocked. Internally KISSAT generates resolvents over the defining clauses for BVE. We modify KISSAT to only detect semantic definition where zero resolvents are generated, ensuring the defining clauses are blocked. We ignore built-in heuristics for selecting candidate variables and instead iterate over all variables.

No nesting information is gathered during definition detection in KISSAT. If a variable is a part of an XOR definition, KISSAT cannot determine if the variable is a defining variable or the definition variable. The defining variables for an XOR may themselves be defined by another definition in the formula. To check for this, if a variable was detected as part of an XOR or semantic definition, the definition clauses were set to inactive and the detection procedure was rerun for that variable.

4 Moving Variables

After all definitions are detected, we move definition variables as close to their defining variables as possible to maximize universal reduction. To do this, we introduce empty existential quantifier levels, denoted T_i, following each $Q_i X_i$ in the prefix yielding

$Q_1X_1\exists T_1 Q_2X_2\exists T_2 \cdots Q_{n-1}X_{n-1}\exists T_{n-1}Q_nX_n$. There is no T_n because variables are not moved inwards. For each definition variable x that can be moved, a fresh variable x' is placed in the quantifier level T_m for $m = \max\{\lambda(z) \mid z \in Z_x\}$. That is, x' will be placed in the existential block that immediately follows the innermost defining variable. Finally, x will be removed from the prefix, and the new formula will be $\psi[x'/x]$.

Example 2. In the formula $\exists x_3 \forall x_1 \exists x_4 \forall x_2 \exists x_5 . (x_5 \vee \overline{x}_4 \vee \overline{x}_3) \wedge (\overline{x}_5 \vee x_3) \wedge (\overline{x}_5 \vee x_4) \wedge (x_5 \vee x_1) \wedge (x_2 \vee x_5)$, the variable x_5 is defined as $x_5 \leftrightarrow x_3 \wedge x_4$, with defining variables $\{x_3, x_4\}$. A fresh variable x_5' is introduced to replace x_5. x_5' is placed in an existential quantifier level following the innermost defining variable x_4. Then, x_5' is substituted for x_5 in the formula giving $\exists x_3 \forall x_1 \exists x_4 \exists x_5' \forall x_2 . (x_5' \vee \overline{x}_4 \vee \overline{x}_3) \wedge (\overline{x}_5' \vee x_3) \wedge (\overline{x}_5' \vee x_4) \wedge (x_5' \vee x_1) \wedge (x_2 \vee x_5')$. Finally, x_2 can be removed from $(x_2 \vee x_5')$ by universal reduction.

Movement requires new variables because QRAT steps either add or delete clauses and cannot affect the quantifier placement of existing variables. When definitions are added in the checker QRAT-TRIM the new definition variables are placed in a quantifier level based on their defining variables. For a definition variable x, if the innermost defining variable $z \in X_i$ is existentially quantified ($Q_i = \exists$) the definition variable is placed in X_i, and if z is universally quantified ($Q_i = \forall$) the definition variable is placed in the existential level X_{i+1}, So, new definition variables are placed in the desired quantifier level. Because contiguous levels with the same quantifier can be combined, the introduction of T levels does not change the semantics.

4.1 Moving in Order

The tools for definition detection run on CNF instances, so, some definitions may not be left-total when considering the prefix. This can occur if the definition variable is in a level outer to one of its defining variables. Also, some monotonic definitions may not satisfy the one-sided property. These problems are checked during proof generation. If they occur, that variable is not moved.

The variable movement algorithm starts at the outermost quantifier level and sweeps inwards, at each step moving all possible definition variables to the current level. A definition variable x can be moved if $x >_\Pi z$ for all $z \in Z_x$, and x is not universally quantified. It can be moved to T_m where $m = \max\{\lambda(z) \mid z \in Z_x\}$, and will be moved during iteration m of the algorithm. A look up table is used to efficiently find definitions with the innermost defining variable at level m. Once a definition variable has been moved, if it was a defining variable for some other definitions, those definitions are checked for movement and the look up table is updated. Since the iteration starts at the outermost level, it guarantees variables that can be moved within our framework are moved as far as possible. This requires a single pass, so moved definitions will not be revisited.

4.2 XOR Processing

In an XOR definition multiple variables are left-total and right-unique. Additional information is required to determine which variable is the proper candidate for movement.

If a variable is defined elsewhere and appears in an XOR, it must be a defining variable in the XOR. In addition, universal variables must be defining variables. However, a distinction cannot be made between the remaining variables before beginning movement.

Example 3. Given the QBF, $\exists_1 x_1, x_2 \forall y_1 \exists_2 x_3 \forall y_2 \exists_3 x_4 \forall y_3 \exists_4 x_5 \forall y_4 \exists_5 x_6, x_7.(x_6 \leftrightarrow x_1 \land x_i) \land (x_3 \oplus x_4 \oplus x_5) \land (x_1 \oplus x_5 \oplus x_6) \land \ldots$, determining the definition variables for the XOR definitions will hinge on the movement of x_6. **Case 1**, Let $x_i = x_7$ in the AND definition, x_6 cannot be moved. Then, x_5 can be moved to $\exists_3$ as the definition variable of $(x_3 \oplus x_4 \oplus x_5)$. No other variables can be moved. **Case 2**, Let $x_i = x_2$ in the AND definitions, x_6 can be moved to $\exists_1$. Then, x_5 can be moved to $\exists_1$ as the definition variable of $(x_1 \oplus x_5 \oplus x_6)$. Next, x_4 can be moved to $\exists_2$ as the definition variable of $(x_3 \oplus x_4 \oplus x_5)$. The possible movement of x_6 will determine how the XOR definitions are moved. This information is not known until runtime, so the definition variable of an XOR cannot be determined before variable movement is performed.

As seen in the example, movement of definition variables can affect what variable in an XOR is eventually moved. The definition variable for an XOR must be determined during the movement process. The definition variable is initially set as the innermost variable in the XOR. If that variable is defined elsewhere and moved, the definition variable of the XOR is reset to the new innermost variable. We perform the same check as the general case to see if the definition variable can be moved. With XOR definitions, the algorithm is still deterministic and produces optimal movement, since all variables that can be moved are moved to their outermost level.

4.3 Proving Variable Movement

In this section we describe how to modify a formula through a series of QRAT clause additions and deletions to achieve variable movement. Moving a definition variable x in the formula $\Pi.\psi$ involves:

- Introducing a new definition variable x' to replace x.
- Deriving an equivalence between x' and x.
- Transforming the formula ψ to $\psi[x'/x]$ with x removed from Π and x' placed in the existential quantifier level following its innermost defining variable.

The algorithm for moving a definition variable x proceeds in five steps, each involving some clause additions or deletions. Some of the steps can be simplified depending on the type of definition. Moving a one-sided definition requires slight modifications to a few steps, and these are discussed following each of the relevant steps.

1. *Add the defining clauses $\delta(x')$ and $\delta(\overline{x}')$.*
   We introduce a fresh existential variable x' and add the defining clauses $\delta(x)[x'/x]$ and $\delta(\overline{x})[x'/x]$. Each clause is Q-blocked on x' or $\overline{x}'$ since the definition is left-total and variable x' is in the quantifier level following its innermost defining variable.
2. *Add the equivalence clauses $x \leftrightarrow x'$.*
   Both x and x' are fully defined by the same set of variables, so it is possible to derive the equivalence clauses $(\overline{x} \lor x')$ and $(x \lor \overline{x}')$. The first implication added

is Q-blocked-subsumed. Consider $(\overline{x} \vee x')$, for each clause $C' \in \delta(\overline{x}')$. The outer resolvent of C' with $(\overline{x} \vee x')$ on x is subsumed by the corresponding $C \in \delta(\overline{x})$. This is not the case for $(x \vee \overline{x}')$ because the outer resolvent of $(x \vee \overline{x}')$ with $(\overline{x} \vee x')$ is not subsumed by the formula. The clause $(x \vee \overline{x}')$ is QRAT for certain definitions, in particular AND/OR. In the general case we generate a chain of Q-resolutions that imply $(x \vee \overline{x}')$. We use defining variable elimination to eliminate Z_x from the formula $\delta(x) \cup \delta(\overline{x}')$. The procedure produces the clause $(x \vee \overline{x}')$. The resolution tree rooted at $(x \vee \overline{x}')$ is traversed in post-order giving the list of clauses $C_1, ..., C_n, (x \vee \overline{x}')$. We add the clauses in order, deriving $(x \vee \overline{x}')$. The clauses are subsumed by $(x \vee \overline{x}')$ and deleted. If defining variable elimination does not produce $(x \vee \overline{x}')$, then the definition is not right-unique. The variable x cannot be moved in this case.
ONE-SIDED: assuming for the one-sided definition that x occurs positively in the defining clauses, the implication $(\overline{x}' \vee x)$ is added. The implication is Q-blocked-subsumed for the same reasons as the first implication above. If x occurs negatively the implication $(\overline{x} \vee x')$ is added. We will continue the remaining steps under the assumption that x occurs positively in the defining clauses for the one-sided case.

3. *Add and remove the remaining clauses $\rho(x)$ and $\rho(\overline{x})$.*
For all clauses $C \in \rho(x)$, $C' \in \rho(x')$ is the Q-resolvent of C with $(\overline{x} \vee x')$ on pivot x, so C' can be added. C can be deleted because it is the Q-resolvent of C' with $(\overline{x}' \vee x)$ on pivot x'. Similar reasoning is used for $C \in \rho(\overline{x})$.
ONE-SIDED: All $C' \in \rho(x')$ are added with the same reasoning as above. However, there is no $(\overline{x} \vee x')$ so $C \in \rho(\overline{x})$ cannot be deleted until step 5.
4. *Remove the equivalence clauses $x \leftrightarrow x'$*
Equivalence clauses $(x \vee \overline{x}'), (\overline{x} \vee x')$ are deleted. $(x \vee \overline{x}')$ is Q-blocked-subsumed on variable x since for all $D \in \delta(\overline{x})$, the outer resolvent of $(x \vee \overline{x}')$ and D is subsumed by the defining clause $D' \in \delta(\overline{x}')$, and the outer resolvent of $(x \vee \overline{x}')$ with $(\overline{x} \vee x')$ is a tautology. Similarly, $(\overline{x} \vee x')$ is Q-blocked-subsumed.
ONE-SIDED: the definition clauses need the implication in order to be deleted, and so deletion is deferred to step 5.
5. *Remove the defining clauses $\delta(x)$ and $\delta(\bar{x})$.*
The defining clauses on x are all Q-blocked and are deleted.
ONE-SIDED: The defining clauses $D \in \delta(x)$ can be deleted because they are Q-resolvents of $D' \in \delta(x')$ with $(\overline{x}' \vee x)$ on x'. Now the clauses $(\overline{x}' \vee x)$ and $\rho(\overline{x})$ are Q-blocked on x because x only occurs negatively. They are deleted.

Given the QBF $\Pi.\psi$, applying the transformation sequentially with definition variables $x_1, \ldots, x_n$ will yield the QBF $\Pi.\psi'$ where all definition variables x_i have been replaced by new variables x_i' and the new variables are in the appropriate quantifier levels. The concatenated series of clause additions and deletions generated for each definition variable gives a QRAT proof of the equivalence between $\Pi.\psi$ and $\Pi.\psi'$

The steps above can also be used to move a definition variable to some existential quantifier between the variable and its innermost defining variable. In addition, a definition variable that is inside its defining variables can be moved further inwards by reversing the steps, but it is not clear when this would be useful.

Example 4. Given the QBF $\exists x_1 \forall x_2.(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2)$, we have the definition $x_1 \leftrightarrow x_2$. The definition is right-unique but the defining clauses are not Q-blocked on

x_1 since x_1 is at an outer quantifier level. The QBF is false but moving x_1 inward would make it true. To avoid this, we only move variables outward.

Example 5. Given the definition $x_1 \oplus x_2 \oplus x_3$ with x_1 as the definition variable we have $\delta(x_1) = \{(x_1 \vee x_2 \vee x_3), (x_1 \vee \overline{x}_2 \vee \overline{x}_3)\}$ and $\delta(\overline{x}'_1) = \{(\overline{x}'_1 \vee \overline{x}_2 \vee x_3), (\overline{x}'_1 \vee x_2 \vee \overline{x}_3)\}$. Defining variable elimination will perform the following steps:

$$\begin{aligned}
\text{Eliminate } x_2 :& \{(x_1 \vee x_2 \vee x_3) \otimes_{x_2} (\overline{x}'_1 \vee \overline{x}_2 \vee x_3), (\overline{x}'_1 \vee x_2 \vee \overline{x}_3) \otimes_{x_2} (x_1 \vee \overline{x}_2 \vee \overline{x}_3)\} \\
& \theta_1 = \{(x_1 \vee \overline{x}'_1 \vee x_3), (x_1 \vee \overline{x}'_1 \vee \overline{x}_3)\} \\
\text{Eliminate } x_3 :& \{(x_1 \vee \overline{x}'_1 \vee x_3) \otimes_{x_3} (x_1 \vee \overline{x}'_1 \vee \overline{x}_3)\} \\
& \theta_2 = \{(x_1 \vee \overline{x}'_1)\}
\end{aligned}$$

The clause additions to derive the second implication in step 2 would be $(x_1 \vee \overline{x}'_1 \vee x_3)$, $(x_1 \vee \overline{x}'_1 \vee \overline{x}_3)$, $(x_1 \vee \overline{x}'_1)$. Each subsequent clause in the list is implied by Q-resolution. With more defining variables, the resolution tree becomes more complex. The derivation will be of the form $\theta'_1, ..., \theta'_{n-1}$ for $\theta'_i \subset \theta_i$ where θ'_i will include only the clauses needed to derive $(x_1 \vee \overline{x}'_1)$. These can be determined by working through the resolution chain backwards from $(x_1 \vee \overline{x}'_1)$.

Example 6. Given the formula $\exists x_1 x_2 x_3 \forall x_5 x_6 \exists x_4 (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_4) \wedge (\overline{x}_2 \vee \overline{x}_4) \wedge (\overline{x}_3 \vee \overline{x}_4) \wedge (x_4 \vee \overline{x}_5) \wedge (x_4 \vee \overline{x}_6)$, we show the steps generating the QRAT proof of movement for variable x_4 with the pivot appearing as the first literal in the clause. Clauses following a d are deleted from the formula.

1. $(x'_4 \vee x_1 \vee x_2 \vee x_3), (\overline{x}'_4 \vee \overline{x}_1), (\overline{x}'_4 \vee \overline{x}_2), (\overline{x}'_4 \vee \overline{x}_3)$
2. $(\overline{x}'_4 \vee x_4), (x'_4 \vee \overline{x}_4)$
3. $(x'_4 \vee \overline{x}_5), d(x_4 \vee \overline{x}_5), (x'_4 \vee \overline{x}_6), d(x_4 \vee \overline{x}_6)$
4. $d(x_4 \vee \overline{x}'_4), d(\overline{x}_4 \vee x'_4)$
5. $d(x_4 \vee x_1 \vee x_2 \vee x_3), d(\overline{x}_4 \vee \overline{x}_1), d(\overline{x}_4 \vee \overline{x}_2), d(\overline{x}_4 \vee \overline{x}_3)$

The definition variable x_4 is replaced by the fresh variable x'_4 which will be placed in the prenex as $\exists x_1 x_2 x_3 \exists x'_4 \forall x_5 x_6$ achieving the desired movement. The QRAT proof system uses a stronger redundancy notion that avoids auxiliary clauses for an AND definition in step 2.

We verified all instances of variable movement on QBFEVAL'20 benchmarks using QRAT-TRIM [8]. By default, QRAT-TRIM will check a satisfaction proof with forward checking, verifying the clause deletion steps are correct in the order they appear. A refutation proof is checked with backward checking, verifying the clause addition steps are correct starting at the empty clause and working backwards. It is not known whether the problem is true or false at the variable movement stage, so both clause addition and deletion steps are checked to preserve equivalence. To do this, we modified QRAT-TRIM by adding a DUAL-FORWARD mode that performs a forward check, verifying both clause additions and deletions. We verified several end-to-end proofs for formulas solved by BLOQQER after variable movement. We appended the BLOQQER proof onto the variable movement proof, and verified it against the original formula with QRAT-TRIM. All formulas that BLOQQER solved after movement were verified in this way.

5 Evaluation

Variable movement is evaluated on 494 of the 521 QBFEVAL'20 benchmarks. Two benchmark families were removed due to resource limits preventing proof verification. We compare definition detection tools KISSAT and CNFTOOLS, then evaluate the affect of variable movement on solver performance. We ran our experiments on StarExec [18]. The compute nodes that ran our experiments were Intel Xeon E5 cores with 2.4 GHz, and all experiments ran with 32 GB. The repository with programs and data is archived at https://zenodo.org/record/5733440.

5.1 Evaluating Definition Detection

The tools are given 10 seconds to detect definitions. KISSAT attempts to check each variable, whereas CNFTOOLS will iterate through root clauses until the time limit. Root clause selection is split into *max-var* (mv) and *minimally-unblocked* (mb). We consider all definitions extracted up to a timeout if one occurs. The combined approach takes the union of definitions found in each tool, and each tool is still allotted 10 seconds.

Figure 1 shows the number of definitions found (top) and moved (bottom) compared to the combined approach. The tools do not go above the diagonal in either plot because the combined approach takes a union of found definitions and movement cannot be worsened by additional definitions. For many formulas multiple tools contribute to the combined total, shown by a column of points where none are on the diagonal. There is a noticeable pattern between CNFTOOLS (mb) and (mv) where (mb) performs slightly worse due to the additional time spent computing the minimally-unblocked root clauses. But there are some instances where the minimally-unblocked heuristic finds definitions that lead to more movement. For combined, definitions were found in 493 instances and moved in 157 instances In comparing the plots it is clear that the number of definitions found is not a strict predictor of movement. KISSAT finds a similar number of definitions as CNFTOOLS for many instances but consistently moves more. Table 1 shows the breakdown of definitions found and moved by type, and the AND/OR found more frequently by KISSAT are moved more often.

Table 1 further illuminates the differences between the tools. CNFTOOLS has syntactic definition detection similar to KISSAT for BiEQ, AND/OR, XOR, but fails to move a fraction of the XOR definitions. CNFTOOLS does detect tens of XORs as monotonic definitions with the wrong definition variable, meaning the BFS picked up nested definitions in the wrong direction w.r.t. quantifier levels. But, the reason for the large gap between CNFTOOLS and KISSAT is efficiency. CNFTOOLS does not detect the vast majority of XOR definitions moved by KISSAT within the time limit, and the same is true for the other definitions. KISSAT uses the entire 10 seconds on 11 formulas whereas CNFTOOLS times out on 111 (mv) and 99 (mb). Increasing the timeout for each tool in the combined approach to 50 seconds produces only 780 more moved variables over 2 formulas. It is clear from the bottom plot in Figure 1 that CNFTOOLS contributes to the movement of the combined approach in a handful of cases where KISSAT is not on the diagonal. Combining the output of the tools makes use of KISSAT's speed in detecting many simple definitions and CNFTOOLS's ability to find one-sided definitions using complex heuristics and hierarchical search.

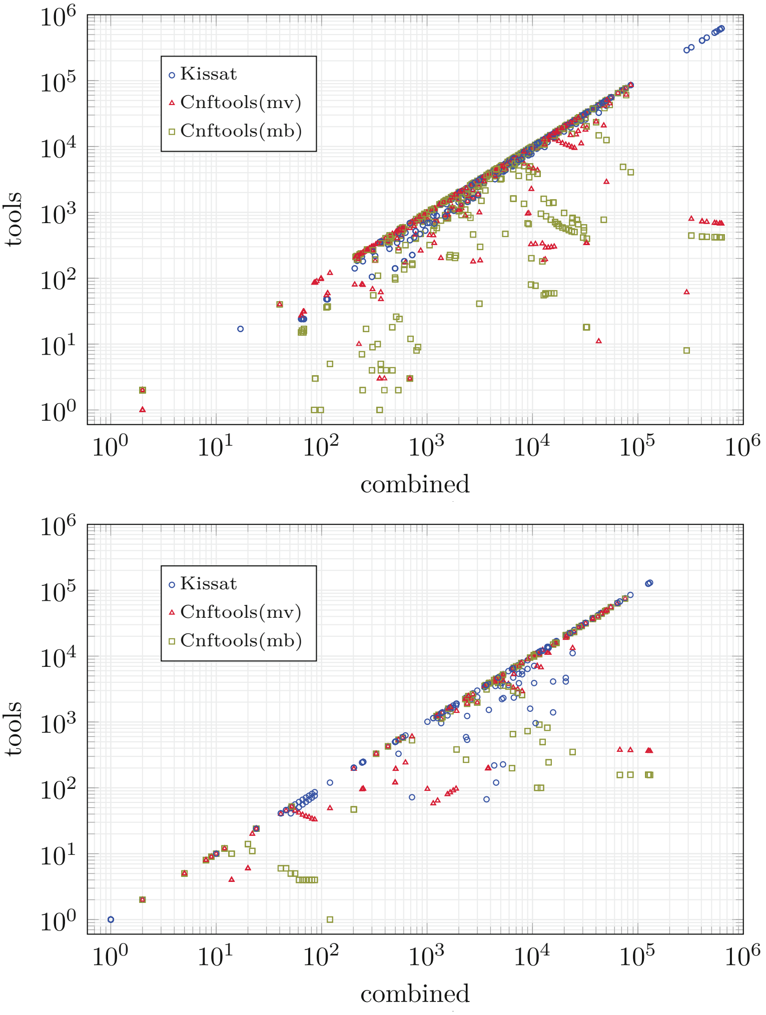


Fig. 1. Comparison of definitions found (top) and moved (bottom) per instance between combined and the individual tools.

No variables found by semantic detection were moved in Kissat and only 88 were moved in Cnftools (mb). Kissat found 159,544 right-unique definitions with Kitten, but only 23,457 were left-total. Of those, the majority had defining variables in the same level as the definition variable, and a smaller fraction had the definition variable at an outer level. For Cnftools 48,715 (mb) and 147,170 (mv) semantic definitions were detected via. right-uniqueness checks. These semantic definitions may not be introduced or manipulated by users in the same way as the standard definitions, explaining why they already occur in the desired quantifier level.

The far most common reason definitions cannot be moved is that they already appear in the same quantifier level as some of their defining variables. For example, many

Table 1. The number of definitions found and moved over all instances. Definitions moved are broken down by a selection of the types, omitting ITE and semantic. Some one-sided definitions CNFTOOLS moves are fully-defined, and combined will move them based on the fully-defined definition provided by KISSAT. So, the missing one-sided definitions for combined are spread across the other definition types.

Detection Tool	Found	Moved	BiEQ	AND/OR	One-Sided	XOR
CNFTOOLS(mv)	3,525,559	1,032,807	21,198	969,630	37,642	0
CNFTOOLS(mb)	2,856,306	935,336	4,619	891,027	39,863	0
KISSAT	9,243,158	1,567,746	308,987	1,215,036	—	42,364
combined	9,624,654	1,664,655	309,793	1,273,381	37,646	42,476

Table 2. The number of definitions found that were not left-total, split by existentially and universally quantified variables, along with monotonic definitions that could not be moved because they were not one-sided. If any universally quantified variable was left-total, the formula would be trivially false.

Detection Tool	Existential	Universal	One-sided
CNFTOOLS(mv)	43,278	11,360	1,107
CNFTOOLS(mb)	23,690	3,771	1,421
KISSAT	32,681	3,219	—

formulas have only two quantifier levels, so there would be no possible movement with all existential variables in the same level. Table 2 shows other reasons a variable may not be moved. A definition is not left-total when the definition variable is at a level outer to some of its defining variables. The tools detected several of these definitions on both universally and existentially quantified variables. Example 2 shows why these variables cannot be moved inwards. Additionally, some of the monotonic definitions extracted by CNFTOOLS are neither fully-defined nor one-sided. These checks are not made until a variable becomes a candidate for movement because a large fraction will be preemptively filtered out due to their quantifier level placement.

CNFTOOLS detect 2,038,407 (mv) and 1,897,482 (mb) monotonic definitions, but this does not match the number of one-sided definitions moved. The majority of monotonic definitions found and moved are actually fully defined. This means for many of the definitions, either $\delta(x)$ or $\delta(\bar{x})$ can be removed from the QBF while preserving equivalence. This can be done in QRAT by recursing through the monotonic definitions and deleting the redundant defining clauses. The large number of fully-defined monotonic definitions shows that QBF formulas generally do not take advantage of optimized encodings, such as the Plaisted-Greenbaum transformation.

5.2 Evaluating Solvers

We used the following solvers to evaluate the impact of variable movement.

- RAREQS (Recursive Abstraction Refinement QBF Solver) [11] pioneered the use of counterexample guided abstraction refinement (CEGAR)-driven recursion and

learning in QBF solvers. The 2012 version has comparable performance to current top-tier solvers.
- CAQE (Clausal Abstraction for Quantifier Elimination) [17] is the first place winner of the 2017, 2018, and 2020 competitions. The solver is written in RUST and based on the CEGAR clausal abstraction algorithm.
- DEPQBF implements the adapted DPLL algorithm QDPLL, relying on dependency schemes to select independent variables for decision making [14]. DEPQBF incorporates QBCE [2] as inprocessing which complicates its relation to preprocessors like BLOQQER.
- GHOSTQ is a non-clausal QBF solver [13]. The solver attempts to convert CNF or QCIR to the GHOSTQ format which introduces Ghost variables, the dual of Tseitin variables. The structural information gained by the conversion is important to GHOSTQ's performance. The conversion relies on the discovery of definitions, which is significantly hampered by preprocessors that delete or change clauses. GHOSTQ also supports a CEGAR extension.

Table 3 shows that variable movement always improves solver performance with and without BLOQQER. Figure 2 provides a more detailed view of the QBF solvers' performance on the original (-o) and moved (-m) formulas using the combined definition detection. The times include definition detection and proof generation, adding 50 seconds on average. In moved formulas, adjacent quantifier levels of the same type were conjoined into a single quantifier level because of GHOSTQ's internal definition detection. This did not impact the other solvers. Movement significantly improves performance of CAQE, DEPQBF, and GHOSTQ-p (plain mode). GHOSTQ-ce (CEGAR mode) and RAREQS improve slightly with movement. Since both GHOSTQ modes use the same conversion to the GHOSTQ format, the impact of variable movement on the conversion does not explain the difference in performance. . Separate experiments moving all definitions except XORs did improve the performance of GHOSTQ in both modes while not affecting other solvers. This is because the conversion to the GHOSTQ format only checks the innermost quantifier level for XOR definitions, and cannot find them if they have been moved. The three solvers implementing CEGAR, GHOSTQ-ce, RAREQS, and CAQE, were affected differently by movement. This may be due to internal heuristics.

Most state-of-the-art QBF solvers make use of preprocessors. The exception is GHOSTQ because its definition detection suffers after the application of QBCE. Fig-

Table 3. The number of instances solved within the 5,000 time-limit over benchmarks where variable movement was possible.

Solver	Original	Moved	BLOQQER	Moved-BLOQQER
CAQE	74	84	99	**103**
GHOSTQ(p)	55	**61**	47	52
GHOSTQ(ce)	77	**80**	65	70
RAREQS	72	72	94	**98**
DEPQBF	64	70	64	**71**

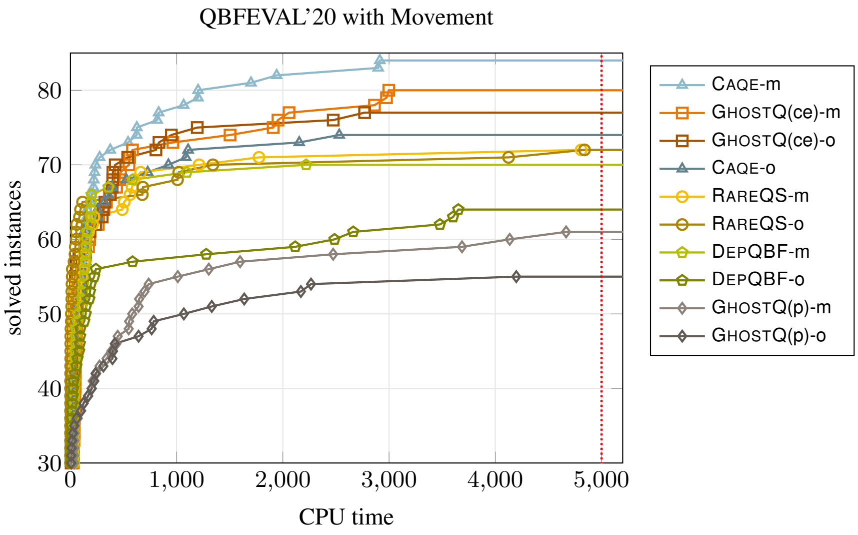


Fig. 2. Cumulative number of solved instances considering only the 157 benchmarks which had variables that could be moved.

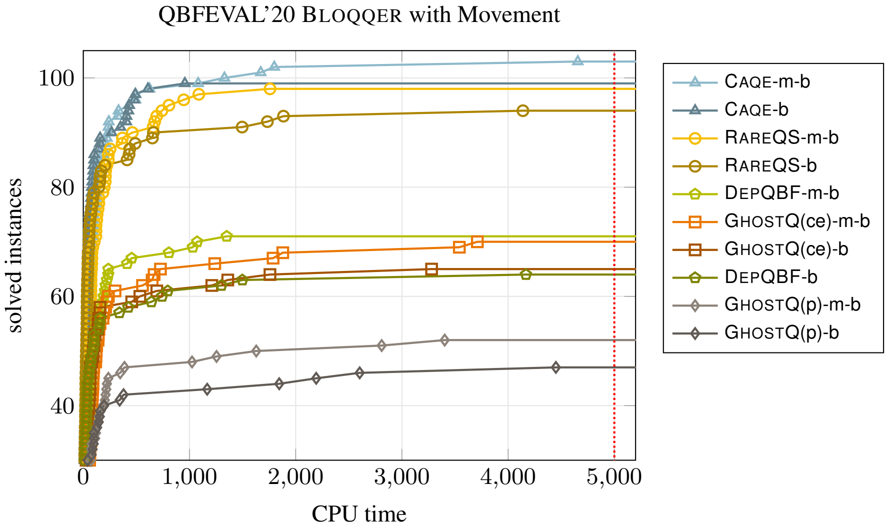


Fig. 3. Cumulative number of solved instances after applying BLOQQER for 100 seconds considering only the 157 benchmarks with movement.

ure 3 shows solver performance with moving variables before applying BLOQQER (m-b) and only applying BLOQQER (-b). The solving time includes the variable movement and BLOQQER runtime within a 100 second timeout. After moving variables, BLOQQER solved 3 formulas and those data are reflected in the plot. In addition, each of the 14 formulas BLOQQER solved before movement, BLOQQER also solved after movement. Performance improved for all solvers when applying variable movement before BLOQQER. One reason for this is movement may allow for more applications of universal reduction. We also experimented with moving variables after BLOQQER preprocessed the formulas. Few variables were moved, and it did not affect solver performance. This is likely due to QBCE removing defining clauses from the formula.

6 PGBDDQ Case Study

Two player games can be succinctly represented in QBF, as an existential player versus a universal opponent. Problem variables encode moves alternating between quantifier levels, and definition variables encode the game state as moves are played over time. Given a $1 \times N$ board, the *linear domino placement* game has two players alternately placing 1×2 dominos on the board. The first player who cannot place a domino loses. The game can be encoded with around $N^2/2$ problem and $3N^2/2$ definition variables.

PGBDDQ is a BDD-based, proof-generating QBF solver. [3] It starts at the innermost quantifier level and performs bucket elimination, linearizing variables and eliminating them through a series of BDD operations that are equivalence-preserving. As BDDs are manipulated, PGBDDQ generates a dual proof through a series of clause additions and deletions. PGBDDQ can solve the linear domino placement problem with polynomial performance when definitions are placed in carefully selected quantifier levels after their defining variables (Manual). In this configuration, moves are processed from the last to the first, with the BDDs at each quantifier level effectively encoding the outcomes of the possible end games for each board state. The performance deteriorates when definition variables are placed in the innermost quantifier level (End).

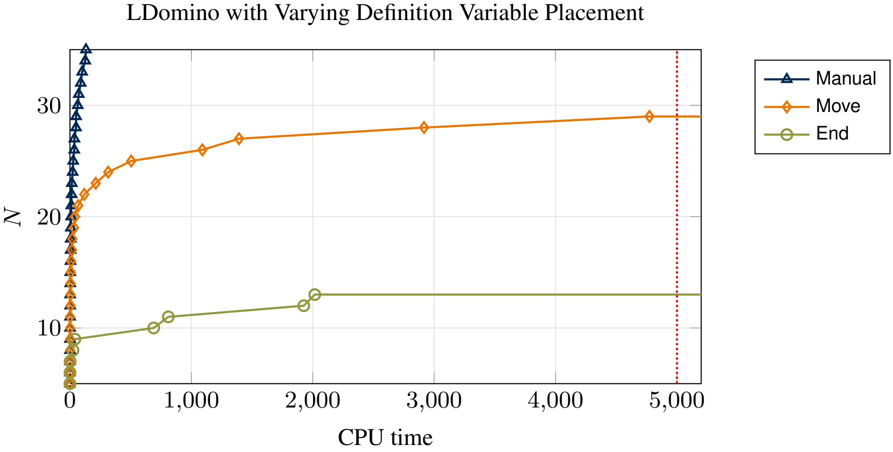


Fig. 4. Performance on boards of size N for false formulas where player two wins. The Move placement times out at $N = 30$ and the End placement runs out of memory at $N = 14$.

In this configuration, the BDDs at each quantifier level must encode the outcomes of the possible end games in terms of the history of all moves up to that point in the game.

Figure 6 shows the performance of PGBDDQ on false formulas where the second player will win. In each configuration, the same hand-crafted BDD variable ordering was used. With the End encoding PGBDDQ runs out of memory on 32 GB RAM at $N = 12$. Applying our movement algorithm to this encoding (Move), the solver performs significantly better and solves all formulas up to $N = 30$ before timeouts occur. This shows how the general problem of memory inefficiency within a BDD can be eased by moving definition variables across quantifier levels. The gap in performance between the Move placement and the Manual placement may be due to the ordering of variables within a quantifier block or moving variables too far outward. When a variable is moved it can be placed anywhere within a quantifier level as this does not change semantics. Also, variables do not need to be moved all the way to their innermost defining variable. Exploring these options in the context of a structurally dependent solver PGBDDQ may lead to improvements that affect other QBF solvers.

7 Conclusion and Future Work

We presented a technique for moving definition variables in QBFs. The movement can be verified within the QRAT proof system, and we validated all proofs in the evaluation with QRAT-TRIM. Using the tools KISSAT and CNFTOOLS to detect definitions, we created a tool-chain for variable movement. On the QBFEVAL'20 benchmarks, one quarter of formulas had definitions that could be moved, and the movement increased solver performance. In addition, we found that movement followed by BLOQQER was more effective than preprocessing with BLOQQER.

For future work, incorporating quantifier level information into definition detection could reduce the costs. For example, the hierarchical detection could recurse outwards based on quantifier levels, reducing the number of root clauses explored and reducing the number of unmoveable definitions detected. Additionaly, there are ways to expand on variable movement. It is possible to place variables anywhere within a given quantifier level and also to adjust how far variables are moved. Optimizing movement may require understanding how variable movement impacts each solver's internal heuristics and solving algorithm. Separately, monotonic definitions that are not one-sided present an interesting challenge for variable movement, as they occur in both polarities outside of the definition. It might also be possible to move the approximately 160,000 semantic definitions found be KITTEN that were right-unique but not left-total.

8 Acknowledgements

We would like to thank Armin Biere for feedback on the semantic definition detection in KISSAT, Markus Iser for insight into the hierarchical definition detection of CNFTOOLS, and Will Klieber for his explanation of GHOSTQ's PCNF converter. We also thank the community at StarExec for providing computational resources.

References

1. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. Tech. rep. (2020)
2. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Automated Deduction (CADE). pp. 101–115. Springer (2011)
3. Bryant, R.E., Heule, M.J.H.: Dual proof generation for quantified Boolean formulas with a BDD-Based solver. In: Automated Deduction (CADE). pp. 433–449. Springer (2021)
4. Davis, M., Putnam, H.: A computing procedure for quantification theory. ACM **7**(3), 394–397 (Jul 1962)
5. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3569, pp. 61–75. Springer (2005)
6. Fleury, M., Biere, A.: Mining definitions in Kissat with Kittens. In: Proceedings of Pragmatics of (SAT) (2021)
7. Heule, M.J.H., Kiesl, B., Biere, A.: Strong extension free proof systems. In: Journal of Automated Reasoning. vol. 64, pp. 533–544 (2020)
8. Heule, M.J.H., Seidl, M., Biere, A.: A unified proof system for QBF preprocessing. In: Automated Reasoning. pp. 91–106. Springer, Cham (2014)
9. Heule, M.J., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: 2013 Formal Methods in Computer-Aided Design (FMCAD). pp. 181–188 (2013)
10. Iser, M.: Recognition and Exploitation of Gate Structure in SAT Solving. Ph.D. thesis, Karlsruhe Institute of Technology (KIT) (2020)
11. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. Artificial Intelligence **234**, 1–25 (2016)
12. Kleine Buning, H., Karpinski, M., Flogel, A.: Resolution for quantified boolean formulas. Inf. Comput. **117**(1), 12–18 (Feb 1995)
13. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A non-prenex, non-clausal QBF solver with game-state learning. In: Theory and Applications of Satisfiability Testing (SAT). pp. 128–142. Springer (2010)
14. Lonsing, F.: Dependency Schemes and Search-Based QBF Solving: Theory and Practice. Ph.D. thesis, Johannes Kepler University (JKU) (2012)
15. Lonsing, F.: QBFRelay, QRATPre+, and DepQBF: Incremental preprocessing meets search-based QBF solving. Journal on Satisfiability, Boolean Modeling and Computation **11**, 211–220 (09 2019)
16. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2**(3), 293–304 (1986)
17. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Formal Methods in Computer-aided Design (FMCAD). pp. 136–143 (September 2015)
18. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 8562, pp. 367–373. Springer (2014)
19. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer (1983)
20. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing (SAT). pp. 422–429. Springer (2014)
21. Wimmer, R., Reimer, S., Marin, P., Becker, B.: HQSpre – an effective preprocessor for QBF and DQBF. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 373–390. Springer (2017)

A Sorted Datalog Hammer for Supervisor Verification Conditions Modulo Simple Linear Arithmetic

Martin Bromberger[1](✉), Irina Dragoste[2], Rasha Faqeh[2], Christof Fetzer[2], Larry González[2], Markus Krötzsch[2], Maximilian Marx[2], Harish K Murali[1,3], and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
{mbromber, weidenb}@mpi-inf.mpg.de
[2] TU Dresden, Dresden, Germany
[3] IIITDM Kancheepuram, Chennai, India

Abstract. In a previous paper, we have shown that clause sets belonging to the Horn Bernays-Schönfinkel fragment over simple linear real arithmetic (HBS(SLR)) can be translated into HBS clause sets over a finite set of first-order constants. The translation preserves validity and satisfiability and it is still applicable if we extend our input with positive universally or existentially quantified verification conditions (conjectures). We call this translation a Datalog hammer. The combination of its implementation in SPASS-SPL with the Datalog reasoner VLog establishes an effective way of deciding verification conditions in the Horn fragment. We verify supervisor code for two examples: a lane change assistant in a car and an electronic control unit of a supercharged combustion engine. In this paper, we improve our Datalog hammer in several ways: we generalize it to mixed real-integer arithmetic and finite first-order sorts; we extend the class of acceptable inequalities beyond variable bounds and positively grounded inequalities; and we significantly reduce the size of the hammer output by a soft typing discipline. We call the result the sorted Datalog hammer. It not only allows us to handle more complex supervisor code and to model already considered supervisor code more concisely, but it also improves our performance on real world benchmark examples. Finally, we replace the before file-based interface between SPASS-SPL and VLog by a close coupling resulting in a single executable binary.

1 Introduction

Modern dynamic dependable systems (e.g., autonomous driving) continuously update software components to fix bugs and to introduce new features. However, the safety requirement of such systems demands software to be safety certified before it can be used, which is typically a lengthy process that hinders the dynamic update of software. We adapt the *continuous certification* approach [17] for variants of safety critical software components using a *supervisor* that guarantees important aspects through *challenging*, see Fig. 1. Specifically, multiple processing units run in parallel – *certified* and *updated not-certified* variants that produce output as *suggestions* and *explications*. The supervisor compares the behavior of variants and analyses their explications. The supervisor itself consists of a rather small set of rules that can be automatically verified and run by a reasoner such as SPASS-SPL. In this paper we concentrate on the further development of our verification approach through the sorted Datalog hammer.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 480–501, 2022.
https://doi.org/10.1007/978-3-030-99524-9_27

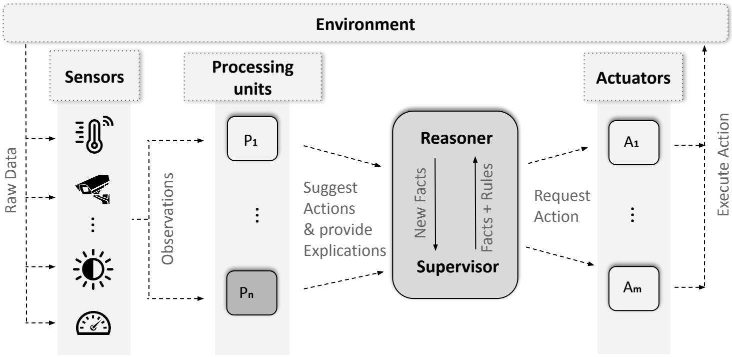


Fig. 1. The supervisor architecture.

While supervisor safety conditions formalized as existentially quantified properties can often already be automatically verified, conjectures about invariants requiring universally quantified properties are a further challenge. Analogous to the Sledgehammer project [8] of Isabelle [31] that translates higher-order logic conjectures to first-order logic (modulo theories) conjectures, our sorted Datalog hammer translates first-order Horn logic modulo arithmetic conjectures into pure Datalog programs, which is equivalent to the Horn Bernays-Schönfinkel clause fragment, called HBS.

More concretely, the underlying logic for both formalizing supervisor behavior and formulating conjectures is the hierarchic combination of the Horn Bernays-Schönfinkel fragment with linear arithmetic, HBS(LA), also called *Superlog* for Supervisor Effective Reasoning Logics [17]. Satisfiability of BS(LA) clause sets is undecidable [15,23], in general, however, the restriction to simple linear arithmetic BS(SLA) yields a decidable fragment [19,22].

Inspired by the test point method for quantifier elimination in arithmetic [27] we show that instantiation with a finite number of values is sufficient to decide whether a universal or existential conjecture is a consequence of a BS(SLA) clause set.

In this paper, we improve our Datalog hammer [11] for HBS(SLA) in three directions. First, we modify our Datalog hammer so it also accepts other sorts for variables besides reals: the integers and arbitrarily many finite first-order sorts $\mathcal{F}_i$. Each non-arithmetic sort has a predefined finite domain corresponding to a set of constants $\mathbb{F}_i$ for $\mathcal{F}_i$ in our signature. Second, we modify our Datalog hammer so it also accepts more general inequalities than simple linear arithmetic allows (but only under certain conditions). In [11], we have already started in this direction by extending the input logic from pure HBS(SLA) to pure positively grounded HBS(SLA). Here we establish a soft typing discipline by efficiently approximating potential values occurring at predicate argument positions of all derivable facts. Third, we modify the test-point scheme that is the basis of our Datalog hammer so it can exploit the fact that not all all inequalities are connected to all predicate argument positions.

Our modifications have three major advantages: first of all, they allow us to express supervisor code for our previous use cases more elegantly and without any additional preprocessing. Second of all, they allow us to formalize supervisor code that would have been out of scope of the logic before. Finally, they reduce the number of required test points, which leads to smaller transformed formulas that can be solved in much less time.

For our experiments of the test point approach we consider again two case studies. First, verification conditions for a supervisor taking care of multiple software variants of a lane change assistant. Second, verification conditions for a supervisor of a supercharged combustion engine, also called an ECU for Electronical Control Unit. The supervisors in both cases are formulated by BS(SLA) Horn clauses. Via our test point technique they are translated together with the verification conditions to Datalog [1] (HBS). The translation is implemented in our Superlog reasoner SPASS-SPL. The resulting Datalog clause set is eventually explored by the Datalog engine VLog [13]. This hammer constitutes a decision procedure for both universal and existential conjectures. The results of our experiments show that we can verify non-trivial existential and universal conjectures in the range of seconds while state-of-the-art solvers cannot solve all problems in reasonable time, see Section 4.

Related Work: Reasoning about BS(LA) clause sets is supported by SMT (Satisfiability Modulo Theories) [30,29]. In general, SMT comprises the combination of a number of theories beyond LA such as arrays, lists, strings, or bit vectors. While SMT is a decision procedure for the BS(LA) ground case, universally quantified variables can be considered by instantiation [36]. Reasoning by instantiation does result in a refutationally complete procedure for BS(SLA), but not in a decision procedure. The Horn fragment HBS(LA) out of BS(LA) is receiving additional attention [20,7], because it is well-suited for software analysis and verification. Research in this direction also goes beyond the theory of LA and considers minimal model semantics in addition, but is restricted to existential conjectures. Other research focuses on universal conjectures, but over non-arithmetic theories, e.g., invariant checking for array-based systems [14] or considers abstract decidability criteria incomparable with the HBS(LA) class [34]. Hierarchic superposition [3] and Simple Clause Learning over Theories (SCL(T)) [12] are both refutationally complete for BS(LA). While SCL(T) can be immediately turned into a decision procedure for even larger fragments than BS(SLA) [12], hierarchic superposition needs to be refined to become a decision procedure already because of the Bernays-Schönfinkel part [21]. Our Datalog hammer translates HBS(SLA) clause sets with both existential and universal conjectures into HBS clause sets which are also subject to first-order theorem proving. Instance generating approaches such as iProver [25] are a decision procedure for this fragment, whereas superposition-based [3] first-order provers such as E [38], SPASS [40], Vampire [37], have additional mechanisms implemented to decide HBS. In our experiments, Section 4, we will discuss the differences between all these approaches on a number of benchmark examples in more detail.

The paper is organized as follows: after a section on preliminaries, Section 2, we present the theory of our sorted Datalog hammer in Section 3, followed by experiments on real world supervisor verification conditions, Section 4. The paper ends with a discussion of the obtained results and directions for future work, Section 5. The artifact (including binaries of our tools and all benchmark problems) is available at [9]. An extended version is available at [10] including proofs and pseudo-code algorithms for the presented results.

2 Preliminaries

We briefly recall the basic logical formalisms and notations we build upon [11]. Starting point is a standard many-sorted first-order language for BS with *constants* (denoted a,b,c), without non-constant function symbols, *variables* (denoted w,x,y,z), and *predicates* (denoted P,Q,R) of some fixed *arity*. *Terms* (denoted t,s) are variables or constants. We write $\bar{x}$ for a vector of variables, $\bar{a}$ for a vector of constants, and so on. An *atom* (denoted A,B) is an expression $P(\bar{t})$ for a predicate P of arity n and a term list $\bar{t}$ of length n. A *positive literal* is an atom A and a *negative literal* is a negated atom $\neg A$. We define $\mathrm{comp}(A) = \neg A$, $\mathrm{comp}(\neg A) = A$, $|A| = A$ and $|\neg A| = A$. Literals are usually denoted L,K,H.

A *clause* is a disjunction of literals, where all variables are assumed to be universally quantified. C,D denote clauses, and N denotes a clause set. We write $\mathrm{atoms}(X)$ for the set of atoms in a clause or clause set X. A clause is *Horn* if it contains at most one positive literal, and a *unit clause* if it has exactly one literal. A clause $A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$ can be written as an implication $B_1 \wedge \ldots \wedge B_m \rightarrow A_1 \vee \ldots \vee A_n$, still omitting universal quantifiers. If Y is a term, formula, or a set thereof, $\mathrm{vars}(Y)$ denotes the set of all variables in Y, and Y is *ground* if $\mathrm{vars}(Y) = \emptyset$. A *fact* is a ground unit clause with a positive literal.

Datalog and the Horn Bernays-Schönfinkel Fragment: The *Horn case of the Bernays-Schönfinkel fragment* (HBS) comprises all sets of clauses with at most one positive literal. The more general Bernays-Schönfinkel fragment (BS) in first-order logic allows arbitrary *formulas* over atoms, i.e., arbitrary Boolean connectives and leading existential quantifiers. BS formulas can be polynomially transformed into clause sets with common syntactic transformations while preserving satisfiability and all entailments that do not refer to auxiliary constants and predicates introduced in the transformation [32]. BS theories in our sense are also known as *disjunctive Datalog programs* [16], specifically when written as implications. A HBS clause set is also called a *Datalog program*. Datalog is sometimes viewed as a second-order language. We are only interested in query answering, which can equivalently be viewed as first-order entailment or second-order model checking [1]. Again, it is common to write clauses as implications in this case.

Two types of *conjectures*, i.e., formulas we want to prove as consequences of a clause set, are of particular interest: *universal* conjectures $\forall \bar{x}.\phi$ and *existential* conjectures $\exists \bar{x}.\phi$, where ϕ is a BS formula that only uses variables in $\bar{x}$. We call such a conjecture positive if the formula only uses conjunctions and disjunctions to connect atoms. Positive conjectures are the focus of our Datalog hammer and they have the useful property that they can be transformed to one atom over a fresh predicate symbol by adding some suitable Horn clause definitions to our clause set N [32,11]. This is also the reason why we assume for the rest of the paper that all relevant universal conjectures have the form $\forall \bar{x}.P(\bar{x})$ and existential conjectures the form $\exists \bar{x}.P(\bar{x})$.

A *substitution* σ is a function from variables to terms with a finite domain $\mathrm{dom}(\sigma) = \{x \mid x\sigma \neq x\}$ and codomain $\mathrm{codom}(\sigma) = \{x\sigma \mid x \in \mathrm{dom}(\sigma)\}$. We denote substitutions by σ,δ,ρ. The application of substitutions is often written postfix, as in $x\sigma$, and is homomorphically extended to terms, atoms, literals, clauses, and quantifier-free formulas. A substitution σ is *ground* if $\mathrm{codom}(\sigma)$ is ground. Let Y denote some term, literal, clause, or clause set. σ is a *grounding* for Y if $Y\sigma$ is ground, and $Y\sigma$ is a *ground instance* of Y in this case. We denote by $\mathrm{gnd}(Y)$ the set of all ground instances of Y, and by $\mathrm{gnd}_B(Y)$ the set of all ground instances over a given set of constants B. The *most general unifier* $\mathrm{mgu}(Z_1,Z_2)$ of two terms/atoms/literals Z_1 and Z_2 is defined as usual, and we assume that it does not introduce fresh variables and is idempotent.

We assume a standard many-sorted first-order logic model theory, and write $\mathcal{A} \models \phi$ if an interpretation $\mathcal{A}$ satisfies a first-order formula ϕ. A formula ψ is a logical consequence of ϕ, written $\phi \models \psi$, if $\mathcal{A} \models \psi$ for all $\mathcal{A}$ such that $\mathcal{A} \models \phi$. Sets of clauses are semantically treated as conjunctions of clauses with all variables quantified universally.

BS with Linear Arithmetic: The extension of BS with linear arithmetic both over real and integer variables, BS(LA), is the basis for the formalisms studied in this paper. We extend the standard *many-sorted* first-order logic with finitely many first-order sorts $\mathcal{F}_i$ and with two arithmetic sorts $\mathcal{R}$ for the real numbers and $\mathcal{Z}$ for the integer numbers. The sort $\mathcal{Z}$ is a *subsort* of $\mathcal{R}$. Given a clause set N, the interpretations $\mathcal{A}$ of our sorts are fixed: $\mathcal{R}^{\mathcal{A}} = \mathbb{R}$, $\mathcal{Z}^{\mathcal{A}} = \mathbb{Z}$, and $\mathcal{F}_i^{\mathcal{A}} = \mathbb{F}_i$, i.e., a first-order sort interpretation $\mathbb{F}_i$ consists of the set of constants in N belonging to that sort, or a single constant out of the signature if no such constant occurs. Note that this is not a deviation from standard semantics in our context as for the arithmetic part the canonical domain is considered and for the first-order sorts BS has the finite model property over the occurring constants which is sufficent for refutation-based reasoning. This way first-order constants are distinct values.

Constant symbols, arithmetic function symbols, variables, and predicates are uniquely declared together with *sort* expressions. The unique sort of a constant symbol, variable, predicate, or term is denoted by the function $\mathrm{sort}(Y)$ and we assume all terms, atoms, and formulas to be well-sorted. The sort of predicate P's argument position i is denoted by $\mathrm{sort}(P,i)$. For arithmetic function symbols we consider the minimal sort with respect to the subsort relation between $\mathcal{R}$ and $\mathcal{Z}$. Eventually, we don't consider arithmetic functions here, so the subsort relationship boils down to substitute an integer sort variable or number for a real sort variable.

We assume *pure* input clause sets, which means the only constants of sort $\mathcal{R}$ or $\mathcal{Z}$ are numbers. This means the only constants that we do allow are integer numbers $c \in \mathbb{Z}$ and the constants defining our finite first-order sorts $\mathcal{F}_i$. Satisfiability of pure BS(LA) clause sets is semi-decidable, e.g., using *hierarchic superposition* [3] or *SCL(T)* [12]. Impure BS(LA) is no longer compact and satisfiability becomes undecidable, but it can be made decidable when restricting to ground clause sets [18].

All arithmetic predicates and functions are interpreted in the usual way. An interpretation of BS(LA) coincides with $\mathcal{A}^{\mathrm{LA}}$ on arithmetic predicates and functions, and freely interprets free predicates. For pure clause sets this is well-defined [3]. Logical satisfaction and entailment is defined as usual, and uses similar notation as for BS.

Example 1. The following BS(LA) clause from our ECU case study compares the values of engine speed (Rpm) and pressure (KPa) with entries in an ignition table (IgnTable) to derive the basis of the current ignition value (IgnDeg1):

$$\begin{aligned}&x_1 < 0 \lor x_1 \geq 13 \lor x_2 < 880 \lor x_2 \geq 1100 \lor \neg \mathrm{KPa}(x_3, x_1) \lor \\ &\neg \mathrm{Rpm}(x_4, x_2) \lor \neg \mathrm{IgnTable}(0, 13, 880, 1100, z) \lor \mathrm{IgnDeg1}(x_3, x_4, x_1, x_2, z)\end{aligned} \tag{1}$$

Terms of the two arithmetic sorts are constructed from a set $\mathcal{X}$ of *variables*, the set of integer constants $c \in \mathbb{Z}$, and binary function symbols $+$ and $-$ (written infix). Atoms in BS(LA) are either *first-order atoms* (e.g., $\mathrm{IgnTable}(0, 13, 880, 1100, z)$) or *(linear) arithmetic atoms* (e.g., $x_2 < 880$). Arithmetic atoms may use the predicates $\leq, <, \neq, =, >, \geq$, which are written infix and have the expected fixed interpretation. Predicates used in first-order atoms are called *free*. *First-order literals* and related notation is defined as before. *Arithmetic literals* coincide with arithmetic atoms, since the arithmetic predicates are closed under negation, e.g., $\neg(x_2 \geq 1100) \equiv x_2 < 1100$.

BS(LA) clauses and conjectures are defined as for BS but using BS(LA) atoms. We often write Horn clauses in the form $\Lambda \,\|\, \Delta \rightarrow H$ where Δ is a multiset of free first-order atoms, H is either a first-order atom or $\bot$, and Λ is a multiset of LA atoms. The semantics of a clause in the form $\Lambda \,\|\, \Delta \rightarrow H$ is $\bigvee_{\lambda \in \Lambda} \neg \lambda \vee \bigvee_{A \in \Delta} \neg A \vee H$, e.g., the clause $x > 1 \vee y \neq 5 \vee \neg Q(x) \vee R(x,y)$ is also written $x \leq 1, y = 5 \| Q(x) \rightarrow R(x,y)$.

A clause or clause set is *abstracted* if its first-order literals contain only variables or first-order constants. Every clause C is equivalent to an abstracted clause that is obtained by replacing each non-variable arithmetic term t that occurs in a first-order atom by a fresh variable x while adding an arithmetic atom $x \neq t$ to C. We asssume abstracted clauses for theory development, but we prefer non-abstracted clauses in examples for readability,e.g., a fact $P(3,5)$ is considered in the development of the theory as the clause $x = 3, x = 5 \,\| \rightarrow P(x,y)$, this is important when collecting the necessary test points. Moreover, we assume that all variables in the theory part of a clause also appear in the first order part, i.e., $\text{vars}(\Lambda) \subseteq \text{vars}(\Delta \rightarrow H)$ for every clause $\Lambda \,\|\, \Delta \rightarrow H$. If this is not the case for x in $\Lambda \,\|\, \Delta \rightarrow H$, then we can easily fix this by first introducing a fresh unary predicate Q over the $\text{sort}(x)$, then adding the literal $Q(x)$ to Δ, and finally adding a clause $\| \rightarrow Q(x)$ to our clause set. Alternatively, x could be eliminated by LA variable elimintation in our context, however this results in a worst case exponential blow up in size. This restriction is necessary because we base all our computations for the test-point scheme on predicate argument positions and would not get any test points for variables that are not connected to any predicate argument positions.

Simpler Forms of Linear Arithmetic: The main logic studied in this paper is obtained by restricting HBS(LA) to a simpler form of linear arithmetic. We first introduce a simpler logic HBS(SLA) as a well-known fragment of HBS(LA) for which satisfiability is decidable [19,22], and later present the generalization HBS(LA)PA of this formalism that we will use.

Definition 2. *The* Horn Bernays-Schönfinkel fragment over simple linear arithmetic, HBS(SLA), *is a subset of* HBS(LA) *where all arithmetic atoms are of the form $x \triangleleft c$ or $d \triangleleft c$, such that $c \in \mathbb{Z}$, d is a (possibly free) constant, $x \in \mathcal{X}$, and $\triangleleft \in \{\leq, <, \neq, =, >, \geq\}$.*

Please note that HBS(SLA) clause sets may be unpure due to free first-order constants of an arithmetic sort. Studying unpure fragments is beyond the scope of this paper but they show up in applications as well.

Example 3. The ECU use case leads to HBS(LA) clauses such as

$$\begin{aligned} &x_1 < y_1 \vee x_1 \geq y_2 \vee x_2 < y_3 \vee x_2 \geq y_4 \vee \neg \text{KPa}(x_3, x_1) \vee \\ &\neg \text{Rpm}(x_4, x_2) \vee \neg \text{IgnTable}(y_1, y_2, y_3, y_4, z) \vee \text{IgnDeg1}(x_3, x_4, x_1, x_2, z). \end{aligned} \tag{2}$$

This clause is not in HBS(SLA), e.g., since $x_1 > x_5$ is not allowed in BS(SLA). However, clause (1) of Example 1 is a BS(SLA) clause that is an instance of (2), obtained by the substitution $\{y_1 \mapsto 0, y_2 \mapsto 13, y_3 \mapsto 880, y_4 \mapsto 1100\}$. This grounding will eventually be obtained by resolution on the IgnTable predicate, because it occurs only positively in ground unit facts.

Example 3 shows that HBS(SLA) clauses can sometimes be obtained by instantiation. In fact, for the satisfiability of an HBS(LA) clause set N only those instances of clauses $(\Lambda \,\|\, \Delta \rightarrow H)\sigma$ are *relevant*, for which we can actually derive all ground facts $A \in \Delta\sigma$ by resolution from N. If A cannot be derived from N and N is satisfiable, then there always exists a satisfying interpretation $\mathcal{A}$ that interprets A as false (and thus $(\Lambda \,\|\, \Delta \rightarrow H)\sigma$ as true). Moreover, if those relevant instances can be simplified to HBS(SLA) clauses, then it is possible to extend almost all HBS(SLA) techniques (including our Datalog hammer) to those HBS(LA) clause sets.

In our case resolution means *hierarchic unit resolution*: given a clause $\Lambda_1 \parallel L, \Delta \rightarrow H$ and a unit clause $\Lambda_2 \parallel \rightarrow K$ with $\sigma = \text{mgu}(L, K)$, their *hierarchic resolvent* is $(\Lambda_1, \Lambda_2 \parallel \Delta \rightarrow H)\sigma$. A fact $P(\bar{a})$ is *derivable* from a pure set of HBS(LA) clauses N if there exists a clause $\Lambda \parallel \rightarrow P(\bar{t})$ that (i) is the result of a sequence of unit resolution steps from the clauses in N and (ii) has a grounding σ such that $P(\bar{t})\sigma = P(\bar{a})$ and $\Lambda\sigma$ evaluates to true. If N is satisfiable, then this means that any fact $P(\bar{a})$ derivable from N is true in all satisfiable interpretations of N, i.e., $N \models P(\bar{a})$. We denote the *set of derivable facts* for a predicate P from N by $\text{dfacts}(P, N)$. A *refutation* is the sequence of resolution steps that produces a clause $\Lambda \parallel \rightarrow \bot$ with $\mathcal{A}^{\text{LA}} \models \Lambda\delta$ for some grounding δ. *Hierarchic unit resolution* is sound and refutationally complete for pure HBS(LA), since every set N of pure HBS(LA) clauses N is *sufficiently complete* [3], and hence *hierarchic superposition* is sound and refutationally complete for N [3,6].

So naturally if all derivable facts of a predicate P already appear in N, then only those instances of clauses can be relevant whose occurrences of P match those facts (i.e., can be resolved with them). We call predicates with this property positively grounded:

Definition 4 (Positively Grounded Predicate [11]). *Let N be a set of* HBS(LA) *clauses. A free first-order predicate P is a* positively grounded predicate *in N if all positive occurrences of P in N are in ground unit clauses (also called facts).*

Definition 5 (Positively Grounded HBS(SLA)**:** HBS(SLA)P **[11]).** *An* HBS(*LA*) *clause set N is out of the fragment* positively grounded HBS(SLA) (HBS(SLA) P) *if we can transform N into an* HBS(SLA) *clause set N′ by first resolving away all negative occurrences of positively grounded predicates P in N, simplifying the thus instantiated* LA *atoms, and finally eliminating all clauses where those predicates occur negatively.*

As mentioned before, if all relevant instances of an HBS(LA) clause set can be simplified to HBS(SLA) clauses, then it is possible to extend almost all HBS(SLA) techniques (including our Datalog hammer) to those clause sets. HBS(SLA)P clause sets have this property and this is the reason, why we managed to extend our Datalog hammer to pure HBS(SLA)P clause sets in [11]. For instance, the set $N = \{P(1), P(2), Q(0), (x \leq y+z \parallel P(y), Q(z) \rightarrow R(x,y))\}$ is an HBS(LA) clause set, but not an HBS(SLA) clause set due to the inequality $x \leq y+z$. Note, however, that the predicates P and Q are positively grounded, the only positive occurrences of P and Q are the facts $P(1)$, $P(2)$, and $Q(0)$. If we resolve with the facts for P and Q and simplify, then we get the clause set $N' = \{P(1), P(2), Q(0), (x \leq 1 \parallel \rightarrow R(x,1)), (x \leq 2 \parallel \rightarrow R(x,2))\}$, which does now belong to HBS(SLA). This means N is a positively grounded HBS(SLA) clause set and our Datalog hammer can still handle it.

Positively grounded predicates are only one way to filter out irrelevant clause instances. As part of our improvements, we define in Section 3 a new logic called approximately grounded HBS(SLA) (HBS(SLA)PA) that is an extension of HBS(SLA)P and serves as the new input logic of our sorted Datalog hammer.

Test-Point Schemes and Functions The Datalog hammer in [11] is based on the following idea: For any pure HBS(SLA) clause set N that is unsatisfiable, we only need to look at the instances $\text{gnd}_B(N)$ of N over finitely many test points B to construct a refutation. Symmetrically, if N is satisfiable, then we can extrapolate a satisfying interpretation for N from a satisfying interpretation for $\text{gnd}_B(N)$. If we can compute such a set of test points B for a clause set N, then we can transform the clause set into an equisatisfiable Datalog program. There exist similar properties for universal/existential conjectures. A *test-point scheme* is an algorithm that can compute such a set of test points B for any HBS(SLA) clause set N and any conjecture $N \models Q\bar{x}.P(\bar{x})$ with $Q \in \{\exists,\forall\}$.

The test-point scheme used by our original Datalog hammer computes the same set of test points for all variables and predicate argument positions. This has several disadvantages: (i) it cannot handle variables with different sorts and (ii) it often selects too many test points (per argument position) because it cannot recognize which inequalities and which argument positions are connected. The goal of this paper is to resolve these issues. However, this also means that we have to assign different test-point sets to different predicate argument positions. We do this with so-called test-point functions.

A *test-point function* (tp-function) β is a function that assigns to some argument positions i of some predicates P a set of test points $\beta(P,i)$. An argument position (P,i) is assigned a set of test points if $\beta(P,i) \subseteq \mathrm{sort}(P,i)^{\mathcal{A}}$ and otherwise $\beta(P,i) = \bot$. A test-point function β is *total* if all argument positions (P,i) are assigned, i.e., $\beta(P,i) \neq \bot$.

A variable x of a clause $\Lambda \parallel \Delta \rightarrow H$ occurs in an argument position (P,i) if $(P,i) \in \mathrm{depend}(x,\Lambda \parallel \Delta \rightarrow H)$, where $\mathrm{depend}(x,Y) = \{(P,i) \mid P(\bar{t}) \in \mathrm{atoms}(Y) \text{ and } t_i = x\}$. Similarly, a variable x of an atom $Q(\bar{t})$ occurs in an argument position (Q,i) if $(Q,i) \in \mathrm{depend}(x,Q(\bar{t}))$. A substitution σ for a clause Y or atom Y is a *well-typed instance* over a tp-function β if it guarantees for each variable x that $x\sigma$ is an element of $\mathrm{sort}(x)^{\mathcal{A}}$ and part of every test-point set (i.e., $x\sigma \in \beta(P,i)$) of every argument position (P,i) it occurs in (i.e., $(P,i) \in \mathrm{depend}(x,Y)$) and that is assigned a test-point set by β (i.e., $\beta(P,i) \neq \bot$). To abbreviate this, we define a set $\mathrm{wti}(x,Y,\beta)$ that contains all values with which a variable can fulfill the above condition, i.e., $\mathrm{wti}(x,Y,\beta) = \mathrm{sort}(x)^{\mathcal{A}} \cap (\bigcap_{(P,i) \in \mathrm{depend}(x,Y) \text{ and } \beta(P,i) \neq \bot} \beta(P,i))$. Following this definition, we denote by $\mathrm{wtis}_{\beta}(Y)$ the *set of all well-typed instances* for a clause/atom Y over the tp-function β, or formally: $\mathrm{wtis}_{\beta}(Y) = \{\sigma \mid \forall x \in \mathrm{vars}(Y).(x\sigma) \in \mathrm{wti}(x,Y,\beta)\}$. With the function gnd_{β}, we denote the *set of all well-typed ground instances* of a clause/atom Y over the tp-function β, i.e., $\mathrm{gnd}_{\beta}(Y) = \{Y\sigma \mid \sigma \in \mathrm{wtis}_{\beta}(Y)\}$, or a set of clauses N, i.e., $\mathrm{gnd}_{\beta}(N) = \{Y\sigma \mid Y \in N \text{ and } \sigma \in \mathrm{wtis}_{\beta}(Y)\}$.

The most general tp-function, denoted by β^*, assigns each argument position to the interpretation of its sort, i.e., $\beta^*(P,i) = \mathrm{sort}(P,i)^{\mathcal{A}}$. So depending on the sort of (P,i), either to $\mathbb{R}$, $\mathbb{Z}$, or one of the $\mathbb{F}_i$. A set of clauses N is satisfiable if and only if $\mathrm{gnd}_{\beta^*}(N)$, the set of all ground instances of N over the base sorts, is satisfiable. Since β^* is the most general tp-function, we also write $\mathrm{gnd}(Y)$ for $\mathrm{gnd}_{\beta^*}(Y)$ and $\mathrm{wtis}(Y)$ for $\mathrm{wtis}_{\beta^*}(Y)$.

If we restrict ourselves to test points, then we also only get interpretations over test points and not for the full base sorts. In order to extrapolate an interpretation from test points to their full sorts, we define extrapolation functions (ep-functions) η. An *extrapolation function* (ep-function) $\eta(P,\bar{a})$ maps an argument vector of test points for predicate P (with $a_i \in \beta(P,i)$) to the subset of $\mathrm{sort}(P,1)^{\mathcal{A}} \times \ldots \times \mathrm{sort}(P,n)^{\mathcal{A}}$ that is supposed to be interpreted the same as $\bar{a}$, i.e., $P(\bar{a})$ is interpreted as true if and only if $P(\bar{b})$ with $\bar{b} \in \eta(P,\bar{a})$ is interpreted as true. By default, any argument vector of test points $\bar{a}$ for P must also be an element of $\eta(P,\bar{a})$, i.e., $\bar{a} \in \eta(P,\bar{a})$. An extrapolation function does not have to be complete for all argument positions, i.e., there may exist argument positions from which we cannot extrapolate to all argument vectors. Formally this means that the actual set of values that can be extrapolated from (P,i) (i.e., $\bigcup_{a_1 \in \beta(P,1)} \ldots \bigcup_{a_n \in \beta(P,n)} \eta(P,\bar{a})$) may be a strict subset of $\mathrm{sort}(P,1)^{\mathcal{A}} \times \ldots \times \mathrm{sort}(P,n)^{\mathcal{A}}$. For all other values $\bar{a}$, $P(\bar{a})$ is supposed to be interpreted as false.

Covering Clause Sets and Conjectures Our goal is to create total tp-functions that restrict our solution space from the infinite reals and integers to finite sets of test points while still preserving (un)satisfiability. Based on these tp-functions, we are then able to define a Datalog hammer that transforms a clause set belonging to (an extension of) HBS(LA) into an equisatisfiable HBS clause set; even modulo universal and existential conjectures.

To be more precise, we are interested in finite tp-functions (together with matching ep-functions) that cover a clause set N or a conjecture $N \models Q\bar{x}.P(\bar{x})$ with $Q \in \{\exists, \forall\}$. A total tp-function β is *finite* if each argument position is assigned to a finite set of test points, i.e., $|\beta(P,i)| \in \mathbb{N}$. A tp-function β *covers a set of clauses* N if $\text{gnd}_\beta(N)$ is equisatisfiable to N. A tp-function β *covers a universal conjecture* $\forall\bar{x}.Q(\bar{x})$ over N if $\text{gnd}_\beta(N) \cup N_Q$ is satisfiable if and only if $N \models \forall\bar{x}.Q(\bar{x})$ is false. Here N_Q is the set $\{\| \text{gnd}_\beta(Q(\bar{x})) \rightarrow \bot\}$ if η is complete for Q or the empty set otherwise. A tp-function β *covers an existential conjecture* $N \models \exists\bar{x}.Q(\bar{x})$ if $\text{gnd}_\beta(N) \cup \text{gnd}_\beta(\| Q(\bar{x}) \rightarrow \bot)$ is satisfiable if and only if $N \models \exists\bar{x}.Q(\bar{x})$ is false.

The most general tp-function β^* obviously covers all HBS(LA) clause sets and conjectures because satisfiability of N is defined over $\text{gnd}_{\beta^*}(N)$. However, β^* is not finite. The test-point scheme in [11], which assigns one finite set of test points B to all variables, also covers clause sets and universal/existential conjectures; at least if we restrict our input to variables over the reals. As mentioned before, the goal of this paper is improve this test-point scheme by assigning different test-point sets to different predicate argument positions.

3 The Sorted Datalog Hammer

In this section, we present a transformation that we call the *sorted Datalog hammer*. It transforms any pure HBS(SLA) clause set modulo a conjecture into an HBS clause set. To guide our explanations, we apply each step of the transformation to a simplified example of the electronic control unit use case:

Example 6. An electronic control unit (ECU) of a combustion engine determines actuator operations. For instance, it computes the ignition timings based on a set of input sensors. To this end, it looks up some base factors from static tables and combines them to the actual actuator values through a series of rules.

In our simplified model of an ECU, we only compute one actuator value, the ignition timing, and we only have an engine speed sensor (measuring in Rpm) as our input sensor. Our verification goal, expressed as a universal conjecture, is to confirm, that the ECU computes an ignition timing for all potential input sensor values. Determining completeness of a set of rules, i.e., determining that the rules produce a result for all potential input values, is also our most common application for universal conjectures. The ECU model is encoded as the following pure HBS(LA) clause set N:

$D_1 : \text{SpeedTable}(0,2000,1350)$, $D_2 : \text{SpeedTable}(2000,4000,1600)$,
$D_3 : \text{SpeedTable}(4000,6000,1850)$, $D_4 : \text{SpeedTable}(6000,8000,2100)$,
$C_1 : 0 \leq x_p, x_p < 8000 \,\|\rightarrow \text{Speed}(x_p)$,
$C_2 : x_1 \leq x_p, x_p < x_2 \,\| \,\text{Speed}(x_p), \text{SpeedTable}(x_1,x_2,y) \rightarrow \text{IgnDeg}(x_p,y)$,
$C_3 : \text{IgnDeg}(x_p,z) \rightarrow \text{ResArgs}(x_p)$, $C_4 : \text{ResArgs}(x_p) \rightarrow \text{Conj}(x_p)$,
$C_5 : x_p \geq 8000 \,\|\rightarrow \text{Conj}(x_p)$, $C_6 : x_p < 0 \,\|\rightarrow \text{Conj}(x_p)$,

In this example all variables are real variables. The clauses $D_1 - D_4$ are table entries from which we determine the base factor of our ignition time based on the speed. Semantically, D_1 : SpeedTable(0,2000,1350) states that the base ignition time is 13.5° before dead center if the engine speed lies between 0Rpm and 2000Rpm. The clause C_1 produces all possible input sensor values labeled by the predicate Speed. The clause C_2 determines the ignition timing from the current speed and the table entries. The end result is stored in the predicate $\mathrm{IgnDeg}(x_p,z)$, where z is the resulting ignition timing and x_p is the speed that led to this result. The clauses $C_3 - C_6$ are necessary for encoding the verification goal as a universal conjecture over a single atom. In clause C_3, the return value is removed from the result predicate $\mathrm{IgnDeg}(x_p,z)$ because for the conjecture we only need to know that there is a result and not what the result is. Clause C_4 guarantees that the conjecture predicate $\mathrm{Conj}(x_p)$ is true if the rules can produce a $\mathrm{IgnDeg}(x_p,z)$ for the sensor value. Clauses $C_5\&C_6$ guarantee that the conjecture predicate is true if one of the sensor values is out of bounds. This flattening process can be done automatically using the techniques outlined in [11]. Hence, the ECU computes an ignition timing for all potential input sensor values if the universal conjecture $\forall x_p.\mathrm{Conj}(x_p)$ is entailed by N.

Approximately Grounded Example 6 contains inequalities that go beyond simple variable bounds, e.g., $x_1 \leq x_p$ in C_2. However, it is possible to reduce the example to an HBS(SLA) clause set. As our first step of the sorted Datalog hammer, we explain a way to heuristically determine which HBS(LA) clause sets can be reduced to HBS(SLA) clause sets. Moreover, we show later that we do not have to explicitly perform this reduction but that we can extend our other algorithms to handle this heuristic extension of HBS(SLA) directly.

We start by formulating an extension of positively grounded HBS(SLA) called approximately grounded HBS(SLA). It is based on over-approximating the set of *derivable values* $\mathrm{dvals}(P,i,N) = \{a_i \mid P(\bar{a}) \in \mathrm{dfacts}(P,N)\}$ for each argument position i of each predicate P in N with only finitely many derivable values, i.e., $|\mathrm{dvals}(P,i,N)| \in \mathbb{N}$. These argument positions are also called *finite*. Naturally, all argument positions over first-order sorts $\mathcal{F}$ are finite argument positions. With regard to clause relevance, only those clause instances are relevant, where a finite argument position is instantiated by one of the derivable values. We call a set of clauses N an approximately grounded HBS(SLA) clause set if all relevant instances based on this criterion can be simplified to HBS(SLA) clauses. For instance, the set $N = \{(x \leq 1 \,\|\!\rightarrow P(x,1)), (x > 2 \,\|\!\rightarrow P(x,3)), (x \geq 0 \,\|\!\rightarrow Q(x,0)), (u \leq y+z \,\|\, P(x,y), Q(x,z) \rightarrow R(x,y,z,u))\}$ is an HBS(LA) clause set, but not a (positively grounded) HBS(SLA) clause set due to the inequality $z \leq y+u$ and the lack of positively grounded predicates. However, the argument positions $(P,2)$, $(Q,2)$, $(R,2)$ and $(R,3)$ only have finitely many derivable values $\mathrm{dvals}(P,2,N) = \mathrm{dvals}(R,2,N) = \{1,3\}$ and $\mathrm{dvals}(Q,2,N) = \mathrm{dvals}(R,3,N) = \{0\}$. If we instantiate all occurrences of P and Q over those values, then we get the set $N' = \{(x \leq 1 \,\|\!\rightarrow P(x,1)), (x > 2 \,\|\!\rightarrow P(x,3)), (x \geq 0 \,\|\!\rightarrow Q(x,0)), (u \leq 1 \,\|\, P(x,1), Q(x,0) \rightarrow R(x,1,0,u)), (u \leq 3 \,\|\, P(x,3), Q(x,0) \rightarrow R(x,3,0,u))\}$ that is an HBS(SLA) clause set. This means N is an approximately grounded HBS(SLA) clause set and our extended Datalog hammer can handle it.

Determining the finiteness of a predicate argument position (and all its derivable values) is not trivial. In general, it is as hard as determining the satisfiability of a clause set [10], so in the case of HBS(LA) undecidable [15,23]. This is the reason, why we only over-approximate the derivable values with the following algorithm.

```
DeriveValues$(N)$
for all predicates $P$ and argument positions $i$ for $P$
    avals$(P,i,N) := \emptyset$;
change $:= \top$;
while (change)
    change $:= \bot$;
    for all Horn clauses $\Lambda \,\|\, \Delta \rightarrow P(t_1,...,t_n) \in N$
        for all argument positions $1 \leq i \leq n$ where avals$(P,i,N) \neq \mathbb{R}$
            if $[(t_i = c)$ or $t_i$ is assigned a constant $c$ in $\Lambda$ and $c \notin$ avals$(P,i,N)]$ then
                avals$(P,i,N) :=$ avals$(P,i,N) \cup \{c\}$,change $:= \top$;
            else if $[t_i$ appears in argument positions $(Q_1,k_1),...,(Q_m,k_m)$ in $\Delta$
                    and avals$(P,i,N) \not\supseteq \bigcap_j$avals$(Q_j,k_j,N)$ ] then
                if $[\mathbb{R} \neq \bigcap_j$avals$(Q_j,k_j,N)]$ then
                    avals$(P,i,N) :=$ avals$(P,i,N) \cup \bigcap_j$avals$(Q_j,k_j,N)$,change $:= \top$;
                else
                    avals$(P,i,N) := \mathbb{R}$,change $:= \top$;
```

At the start, DeriveValues(N) sets avals$(P,i,N) = \emptyset$ for all predicate argument positions. Then it repeats iterating over the clauses in N and uses the current sets avals in order to derive new values, until it reaches a fixpoint. Whenever, DeriveValues(N) computes that a clause can derive infinitely many values for an argument position, it simply sets avals$(P,i,N) = \mathbb{R}$ for both real and integer argument positions. This is the case, when we have a clause $\Lambda \,\|\, \Delta \rightarrow P(t_1,...,t_n)$, and an argument position i for P, such that: (i) t_i is not a constant (and therefore a variable), (ii) t_i is not assigned a constant c in Λ (i.e., there is no equation $t_i = c$ in Λ), (iii) t_i is only connected to argument positions $(Q_1,k_1),...,(Q_m,k_m)$ in Δ that already have avals$(Q_j,k_j,N) = \mathbb{R}$. The latter also includes the case that t_i is not connected to any argument positions in Δ. For instance, DeriveValues(N) would recognize that clause C_1 in example 6 can be used to derive infinitely many values for the argument position (Speed,1) because the variable x_p is not assigned an equation in C_1's theory constraint $\Lambda := (0 \leq x_p, x_p < 8000)$ and x_p is not connected to any argument position on the left side of the implication. Hence, DeriveValues(N) would set avals(Speed,1,N) $= \mathbb{R}$.

For each run through the while loop, at least one predicate argument position is set to $\mathbb{R}$ or the set is extended by at least one constant. The set of constants in N as well as the number of predicate argument positions in N are finite, hence DeriveValues(N) terminates. It is correct because in each step it over-approximates the result of a hierarchic unit resulting resolution step, see Section 2. The above algorithm is highly inefficient. In our own implementation, we only apply it if all clauses are non-recursive and by first ordering the clauses based on their dependencies. This guarantees that every clause is visited at most once and is sufficient for both of our use cases.

Based on avals, we can now build a tp-function β^a that maps all finite argument positions (P,i) that our over-approximation detected to the over-approximation of their derivable values, i.e., $\beta^a(P,i) :=$ avals(P,i,N) if $|$avals$(P,i,N)| \in \mathbb{N}$ and $\beta^a(P,i) := \bot$ otherwise. With β^a we derive the finitely grounded over-approximation agnd(Y) of a set of clauses Y, a clause Y or an atom Y. This set is equivalent to gnd$_{\beta^a}(Y)$, except that we assume that all LA atoms are simplified until they contain at most one integer number and that LA atoms that can be evaluated are reduced to true and false and the respective clause simplified. Based of agnd(N) we define a new extension of HBS(SLA) called approximately grounded HBS(SLA):

Definition 7 (Approximately Grounded $\mathrm{HBS(SLA)}$**:** $\mathrm{HBS(SLA)A}$**).** *A clause set N is out of the fragment* approximately grounded $\mathrm{HBS(SLA)}$ *or short* $\mathrm{HBS(SLA)A}$ *if* $\mathrm{agnd}(N)$ *is out of the* $\mathrm{HBS(SLA)}$ *fragment. It is called* $\mathrm{HBS(SLA)PA}$ *if it is also pure.*

Example 8. Executing $\mathrm{DeriveValues}(N)$ on example 6 leads to the following results:
$\mathrm{avals}(\mathrm{SpeedTable},1,N) = \{0,2000,4000,6000\}$,
$\mathrm{avals}(\mathrm{SpeedTable},2,N) = \{2000,4000,6000,8000\}$,
$\mathrm{avals}(\mathrm{SpeedTable},3,N) = \{1350,1600,1850,2100\}$,
$\mathrm{avals}(\mathrm{IgnDeg},2,N) = \{1350,1600,1850,2100\}$,
and all other argument positions (P,i) are infinite so $\mathrm{avals}(P,i,N) = \mathbb{R}$ for them.

We can now easily check whether $\mathrm{agnd}(N)$ would turn our clause set into an $\mathrm{HBS(SLA)}$ fragment by checking whether the following holds for all inequalities: all variables in the inequality except for one must be connected to a finite argument position on the left side of the clause it appears in. This guarantees that all but one variable will be instantiated in $\mathrm{agnd}(N)$ and the inequality can therefore be simplified to a variable bound.

Connecting Argument Positions and Selecting Test Points As our second step, we are reducing the number of test points per predicate argument position by incorporating that not all argument positions are connected to all inequalities. This also means that we select different sets of test points for different argument positions. For finite argument positions, we can simply pick $\mathrm{avals}(P,i,N)$ as its set of test points. However, before we can compute the test-point sets for all other argument positions, we first have to determine to which inequalities and other argument positions they are connected.

Let N be an $\mathrm{HBS(SLA)PA}$ clause set and (P,i) an argument position for a predicate in N. Then we denote by $\mathrm{conArgs}(P,i,N)$ the *set of connected argument positions* and by $\mathrm{conIneqs}(P,i,N)$ the *set of connected inequalities*. Formally, $\mathrm{conArgs}(P,i,N)$ is defined as the minimal set that fulfills the following conditions: (i) two argument positions (P,i) and (Q,j) are connected if they share a variable in a clause in N, i.e., $(Q,j) \in \mathrm{conArgs}(P,i,N)$ if $(\Lambda \parallel \Delta \rightarrow H) \in N$, $P(\bar{t}),Q(\bar{s}) \in \mathrm{atoms}(\Delta \cup \{H\})$, and $t_i = s_j = x$; and (ii) the connection relation is transitive, i.e., if $(Q,j) \in \mathrm{conArgs}(P,i,N)$, then $\mathrm{conArgs}(P,i,N) = \mathrm{conArgs}(Q,j,N)$. Similarly, $\mathrm{conIneqs}(P,i,N)$ is defined as the minimal set that fulfills the following conditions: (i) an argument position (P,i) is connected to an instance λ' of an inequality λ if they share a variable in a clause in N, i.e., $\lambda' \in \mathrm{conIneqs}(P,i,N)$ if $(\Lambda \parallel \Delta \rightarrow H) \in N$, $P(\bar{t}) \in \mathrm{atoms}(\Delta \cup \{H\})$, $t_i = x$, $(\Lambda' \parallel \Delta' \rightarrow H') \in \mathrm{agnd}(\Lambda \parallel \Delta \rightarrow H)$, $\lambda' \in \Lambda'$, and $\lambda' = x \triangleleft c$ (where $\triangleleft = \{<,>,\leq,\geq,=,\neq\}$ and $c \in \mathbb{Z}$); (ii) an argument position (P,i) is connected to a value $c \in \mathbb{Z}$ if $P(\bar{t})$ with $t_i = c$ appears in a clause in N, i.e., $(x = c) \in \mathrm{conIneqs}(P,i,N)$ if $(\Lambda \parallel \Delta \rightarrow H) \in N$, $P(\bar{t}) \in \mathrm{atoms}(\Delta \cup \{H\})$, and $t_i = c$; (iii) an argument position (P,i) is connected to a value $c \in \mathbb{Z}$ if (P,i) is finite and $c \in \mathrm{avals}(P,i,N)$, i.e., $(x = c) \in \mathrm{conIneqs}(P,i,N)$ if (P,i) is finite and $c \in \mathrm{avals}(P,i,N)$; and (iv) the connection relation is transitive, i.e., $\lambda \in \mathrm{conArgs}(Q,j,N)$ if $\lambda \in \mathrm{conIneqs}(P,i,N)$ and $(Q,j) \in \mathrm{conArgs}(P,i,N)$.

Example 9. To highlight the connections in example 6 more clearly, we use the same variable symbol for connected argument positions. Therefore $(\mathrm{SpeedTable},1)$ and $(\mathrm{SpeedTable},2)$ are only connected to themselves and $\mathrm{conArgs}(\mathrm{SpeedTable},3,N) = \{(\mathrm{SpeedTable},3),(\mathrm{IgnDeg},2)\}$, and $\mathrm{conArgs}(\mathrm{Speed},1,N) = \{(\mathrm{Speed},1),(\mathrm{IgnDeg},1),(\mathrm{ResArgs},1),(\mathrm{Conj},1)\}$, Computing the connected argument positions is a little bit more complicated: first, if a connected argument position is finite, then we have to add all values in avals as equations to the connected inequalities. E.g., $\text{conIneqs}(\text{SpeedTable},1,N) = \{x_1 = 0, x_1 = 2000, x_1 = 4000, x_1 = 6000\}$ because $\text{avals}(\text{SpeedTable},1,N) = \{0,2000,4000,6000\}$. Second, we have to add all inequalities connected in $\text{agnd}(N)$. Again this is possible without explicitly computing $\text{agnd}(N)$. E.g., for the inequality $x_1 \leq x_p$ in clause C_2, we determine that x_1 is connected to the finite argument position (SpeedTable,1) in C_2 and x_p is not connected to any finite argument positions. Hence, we have to connect the following variable bounds to all argument positions connected to x_p, i.e., $\{x_1 \leq x_p \mid x_1 \in \text{avals}(\text{SpeedTable},1,N)\} = \{x_p \geq 0, x_p \geq 2000, x_p \geq 4000, x_p \geq 6000\}$ to the argument positions $\text{conArgs}(\text{Speed},1,N)$. If we apply the above two steps to all clauses, then we get as connected inequalities: $\text{conIneqs}(\text{SpeedTable},2,N) = \{x_2 = 2000, x_2 = 4000, x_3 = 6000, x_4 = 8000\}$, $\text{conIneqs}(\text{SpeedTable},3,N) = \{y = 1350, y = 1600, y = 1850, y = 2100\}$, and $\text{conIneqs}(\text{Speed},1,N) = \{x_p < 0, x_p < 2000, x_p < 4000, x_p < 6000, x_p < 8000, x_p \geq 0, x_p \geq 2000, x_p \geq 4000, x_p \geq 6000, x_p \geq 8000\}$.

Now based on these sets we can construct a set of test points as follows: For each argument position (P,i), we partition the reals $\mathbb{R}$ into intervals such that any variable bound in $\lambda \in \text{conIneqs}(P,i,N)$ is satisfied by all points in one such interval I or none. Since we are in the Horn case, this is enough to ensure that we derive facts *uniformly* over those intervals and the integers/non-integers. To be more precise, we derive facts *uniformly* over those intervals and the integers because $P(\bar{a})$ is derivable from N and $a_i \in I \cap \mathbb{Z}$ implies that $P(\bar{b})$ is also derivable from N, where $b_j = a_j$ for $i \neq j$ and $b_i \in I \cap \mathbb{Z}$. Similarly, we derive facts *uniformly* over those intervals and the non-integers because $P(\bar{a})$ is derivable from N and $a_i \in I \setminus \mathbb{Z}$ implies that $P(\bar{b})$ is also derivable from N, where $b_j = a_j$ for $i \neq j$ and $b_i \in I$. As a result, it is enough to pick (if possible) one integer and one non-integer test point per interval to cover the whole clause set.

Formally we compute the interval partition $\text{iPart}(P,i,N)$ and the set of test points $\text{tps}(P,i,N)$ as follows: First we transform all variable bounds $\lambda \in \text{conIneqs}(P,i,N)$ into interval borders. A variable bound $x \triangleleft c$ with $\triangleleft \in \{\leq,<,>,\geq\}$ in $\text{conIneqs}(P,i,N)$ is turned into two interval borders. One of them is the interval border implied by the bound itself and the other its negation, e.g., $x \geq 5$ results in the interval border $[5$ and the interval border of the negation $5)$. Likewise, we turn every variable bound $x \triangleleft c$ with $\triangleleft \in \{=,\neq\}$ into all four possible interval borders for c, i.e. $c)$, $[c,\, c]$, and $(c$. The set of interval borders $\text{iEP}(P,i,N)$ is then defined as follows:

$$\begin{aligned}\text{iEP}(P,i,N) = {} & \{c],(c \mid x \triangleleft c \in \text{conIneqs}(P,i,N) \text{ where } \triangleleft \in \{\leq,=,\neq,>\}\} \cup \\ & \{c),[c \mid x \triangleleft c \in \text{conIneqs}(P,i,N) \text{ where } \triangleleft \in \{\geq,=,\neq,<\}\} \cup \{(-\infty,\infty)\}\end{aligned}$$

The interval partition $\text{iPart}(P,i,N)$ can be constructed by sorting $\text{iEP}(P,i,N)$ in an ascending order such that we first order by the border value—i.e. $\delta < \epsilon$ if $\delta \in \{c),[c,c],(c\}$, $\epsilon \in \{d),[d,d],(d\}$, and $c < d$—and then by the border type—i.e. $c) < [c < c] < (c$. The result is a sequence $[\ldots,\delta_l,\delta_u,\ldots]$, where we always have one lower border δ_l, followed by one upper border δ_u. We can guarantee that an upper border δ_u follows a lower border δ_l because $\text{iEP}(P,i,N)$ always contains $c)$ together with $[c$ and $c]$ together with $(c$ for $c \in \mathbb{Z}$, so always two consecutive upper and lower borders. Together with $(-\infty$ and $\infty)$ this guarantees that the sorted $\text{iEP}(P,i,N)$ has the desired structure. If we combine every two subsequent borders δ_l, δ_u in our sorted sequence $[\ldots,\delta_l,\delta_u,\ldots]$, then we receive our partition of intervals $\text{iPart}(P,i,N)$. For instance, if $x < 5$ and $x = 0$ are the only variable bounds in $\text{conIneqs}(P,i,N)$, then $\text{iEP}(P,i,N) = \{5),[5,0),[0,0],(0,(-\infty,\infty)\}$ and if we sort and combine them we get $\text{iPart}(P,i,N) = \{(-\infty,0),[0,0],(0,5),[5,\infty)\}$.

After constructing $\text{iPart}(P,i,N)$, we can finally construct the set of test points $\text{tps}(P,i,N)$ for argument position (P,i). If $|\text{avals}(P,i,N)| \in \mathbb{N}$, i.e., we determined that (P,i) is finite, then $\text{tps}(P,i,N) = \text{avals}(P,i,N)$. If the argument position (P,i) is over a first-order sort $\mathcal{F}_i$, i.e., $\text{sort}(P,i) = \mathcal{F}_i$, then we should always be able to determine that (P,i) is finite because $\mathbb{F}_i$ is finite. If the argument position (P,i) is over an arithmetic sort, i.e., $\text{sort}(P,i) = \mathcal{R}$ or $\text{sort}(P,i) = \mathcal{Z}$, and our approximation could not determine that (P,i) is finite, then the test-point set $\text{tps}(P,i,N)$ for (P,i) consists of at most two points per interval $I \in \text{iPart}(P,i,N)$: one integer value $a_I \in I \cap \mathbb{Z}$ if I contains integers (i.e. if $I \cap \mathbb{Z} \neq \emptyset$) and one non-integer value $b_I \in I \setminus \mathbb{Z}$ if I contains non-integers (i.e. if I is not just one integer point). Additionally, we enforce that $\text{tps}(P,i,N) = \text{tps}(Q,j,N)$ if $\text{conArgs}(P,i,N) = \text{conArgs}(Q,j,N)$ and both (P,i) and (Q,j) are infinite argument positions. (In our implementation of this test-point scheme, we optimize the test point selection even further by picking only one test point per interval—if possible an integer value and otherwise a non-integer—if all $\text{conArgs}(P,i,N)$ and all variables x connecting them in N have the same sort. However, we do not prove this optimization explicitly here because the proofs are almost identical to the case for two test points per interval.)

Based on these sets, we can now also define a tp-function β and an ep-function η. For the tp-function, we simply assign any argument position to $\text{tps}(P,i,N)$, i.e., $\beta(P,i) = \text{tps}(P,i,N) \cap \text{sort}(P,i)^{\mathcal{A}}$. (The intersection with $\text{sort}(P,i)^{\mathcal{A}}$ is needed to guarantee that the test-point set of an integer argument position is well-typed.) This also means that β is total and finite. For the ep-function η, we extrapolate any test-point vector $\bar{a}$ (with $\bar{a} = \bar{x}\sigma$ and $\sigma \in \text{wtis}_\beta(P(\bar{x}))$) over the (non-)integer subset of the intervals the test points belong to, i.e., $\eta(P,\bar{a}) = I'_1 \times \ldots \times I'_n$, where $I'_i = \{a_i\}$ if we determined that (P,i) is finite and otherwise I_i is the interval $I_i \in \text{iPart}(P,i,N)$ with $a_i \in I_i$ and $I'_i = I_i \cap \mathbb{Z}$ if a_i is an integer value and $I'_i = I_i \setminus \mathbb{Z}$ if a_i is a non-integer value. Note that this means that η might not be complete for every predicate P, e.g., when P has a finite argument position (P,i) with an infinite domain. However, both β and η together still cover the clause set N, cover any universal conjecture $N \models \forall \bar{x}.Q(\bar{x})$, and cover any existential conjecture $N \models \exists \bar{x}.Q(\bar{x})$.

Theorem 10. *The* tp-function β *covers* N*. The* tp-function β *covers an existential conjecture* $N \models \exists \bar{x}.Q(\bar{x})$*. The* tp-function β *covers a universal conjecture* $N \models \forall \bar{x}.Q(\bar{x})$.

Example 11. Continuation of example 6: The majority of argument positions in our example are finite. Hence, determining their test point set is equivalent to the over-approximation of derivable values avals we computed for them: $\beta(\text{SpeedTable},1) = \{0,2000,4000,6000\}$, $\beta(\text{SpeedTable},2) = \{2000,4000,6000,8000\}$, $\beta(\text{SpeedTable},3) = \{1350,1600,1850,2100\}$, and $\beta(\text{IgnDeg},2) = \{1350,1600,1850,2100\}$. The other argument positions are all connected to $(\text{Speed},1)$ and $\text{conIneqs}(\text{Speed},1,N) = \{x_p < 0, x_p < 2000, x_p < 4000, x_p < 6000, x_p < 8000, x_p \geq 0, x_p \geq 2000, x_p \geq 4000, x_p \geq 6000, x_p \geq 8000\}$, from which we can compute

$$\text{iPart}(P,i,N) = \{(-\infty,0),[0,2000),[2000,4000),[4000,6000),[6000,8000),[8000,\infty)\}$$

and select the test point sets $\beta(\text{Speed},1) = \beta(\text{IgnDeg},1) = \beta(\text{ResArgs},1) = \beta(\text{Conj},1) = \{-1,0,2000,4000,6000,8000\}$. (Note that all variables in our problem are over the reals, so we only have to select one test point per interval! Moreover, in our previous version of the test point scheme, there would have been more intervals in the partition because we would have processed all inequalities, e.g., also those in $\text{conIneqs}(\text{SpeedTable},3,N)$.) The ep-function η that determines which interval is represented by which test point is $\eta(P,1,-1) = (-\infty,0)$, $\eta(P,1,0) = [0,2000)$, $\eta(P,1,2000) = [2000,4000)$, $\eta(P,1,4000) = [4000,6000)$, $\eta(P,1,6000) = [6000,8000)$, $\eta(P,1,8000) = [8000,\infty)$ for the predicates Speed, IgnDeg, ResArgs, and Conj. η behaves like the identity function for all other argument positions because they are finite.

From a Test-Point Function to a Datalog Hammer We can use the covering definitions, e.g., $\text{gnd}_\beta(N)$ is equisatisfiable to N, to instantiate our clause set (and conjectures) with numbers. As a result, we can simply evaluate all theory atoms and thus reduce our HBS(SLA)PA clause sets/conjectures to ground HBS clause sets, which means we could reduce our input into formulas without any arithmetic theory that can be solved by any Datalog reasoner. There is, however, one problem. The set $\text{gnd}_\beta(N)$ grows exponentially with regard to the maximum number of variables n_C in any clause in N, i.e. $O(|\text{gnd}_\beta(N)|) = O(|N| \cdot |B|^{n_C})$, where $B = \max_{(P,i)}(\beta(P,i))$ is the largest test-point set for any argument position. Since n_C is large for realistic examples, e.g., in our examples the size of n_C ranges from 9 to 11 variables, the finite abstraction is often too large to be solvable in reasonable time. Due to this blow-up, we have chosen an alternative approach for our Datalog hammer. This hammer exploits the ideas behind the covering definitions and will allow us to make the same ground deductions, but instead of grounding everything, we only need to (i) ground the negated conjecture over our tp-function and (ii) provide a set of ground facts that define which theory atoms are satisfied by our test points. As a result, the hammered formula is much more concise and we need no actual theory reasoning to solve the formula. In fact, we can solve the hammered formula by greedily applying unit resolution until this produces the empty clause—which would mean the conjecture is implied—or until it produces no more new facts—which would mean we have found a counter example. In practice, greedily applying resolution is not the best strategy and we recommend to use more advanced HBS techniques for instance those used by a state-of-the-art Datalog reasoner.

The Datalog hammer takes as input (i) an HBS(SLA)PA clause set N and (ii) optionally a universal conjecture $\forall \bar{y}.P(\bar{y})$. The case for existential conjectures is handled by encoding the conjecture $N \models \exists \bar{x}.Q(\bar{x})$ as the clause set $N \cup \{Q(\bar{x}) \rightarrow \bot\}$, which is unsatisfiable if and only if the conjecture holds. Given this input, the Datalog hammer first computes the tp-function β and the ep-function η as described above. Next, it computes four clause sets that will make up the Datalog formula. The first set $\text{tren}_N(N)$ is computed by abstracting away any arithmetic from the clauses $(\Lambda \,\|\, \Delta \rightarrow H) \in N$. This is done by replacing each theory atom A in Λ with a literal $P_A(\bar{x})$, where $\text{vars}(A) = \text{vars}(\bar{x})$ and P_A is a fresh predicate. The abstraction of the theory atoms is necessary because Datalog does not support non-constant function symbols (e.g., +,−) that would otherwise appear in approximately grounded theory atoms. Moreover, it is necessary to add extra sort literals $\neg Q_{(P,i,S)}(x)$ for some of the variables $x \in \text{vars}(H)$, where $H = P(\bar{t})$, $t_i = x$, $\text{sort}(x) = S$, and $Q_{(P,i,S)}$ is a fresh predicate. This is necessary in order to define the test point set for x if x does not appear in Λ or in Δ. It is also necessary in order to filter out any test points that are not integer values if x is an integer variable (i.e. $\text{sort}(x) = \mathcal{Z}$) but connected only to real sorted argument positions in Δ (i.e. $\text{sort}(Q,j) = \mathcal{R}$ for all $(Q,j) \in \text{depend}(x,\Delta)$). It is possible to reduce the number of fresh predicates needed, e.g., by reusing the same predicate for two theory atoms whose variables range over the same sets of test points. The resulting abstracted clause has then the form $\Delta_T, \Delta_S, \Delta \rightarrow H$, where Δ_T contains the abstracted theory literals (e.g. $P_A(\bar{x}) \in \Delta_T$) and Δ_S the "sort" literals (e.g. $Q_{(P,i,S)}(x) \in \Delta_S$). The second set is denoted by N_C and it is empty if we have no universal conjecture or if η does not cover our conjecture. Otherwise, N_C contains the ground and negated version ϕ of our universal conjecture $\forall \bar{y}.P(\bar{y})$. ϕ has the form $\Delta_\phi \rightarrow \bot$, where $\Delta_\phi = \text{gnd}_\beta(P(\bar{y}))$ contains all literals $P(\bar{y})$ for all groundings over β. We cannot skip this grounding but the worst-case size of Δ_ϕ is $O(\text{gnd}_\beta(P(\bar{y}))) = O(|B|^{n_\phi})$, where $n_\phi = |\bar{y}|$, which is in our applications typically much smaller than the maximum number of variables n_C contained in some clause in N. The third set is denoted by $\text{tfacts}(N,\beta)$ and contains a fact $\text{tren}_N(A)$ for every ground theory atom A contained in the theory part Λ of a clause $(\Lambda \,\|\, \Delta \rightarrow H) \in \text{gnd}_\beta(N)$ such that A simplifies to true. This is enough to ensure that our abstracted theory predicates evaluate every test point in every satisfiable interpretation $\mathcal{A}$ to true that also would have evaluated to true in the actual theory atom. Alternatively, it is also possible to use a set of axioms and a smaller set of facts and let the Datalog reasoner compute all relevant theory facts for itself. The set $\text{tfacts}(N,\beta)$ can be computed without computing $\text{gnd}_\beta(N)$ if we simply iterate over all theory atoms A in all constraints Λ of all clauses $Y = \Lambda \,\|\, \Delta \rightarrow H$ (with $Y \in N$) and compute all well typed groundings $\tau \in \text{wtis}_\beta(Y)$ such that $A\tau$ simplifies to true. This can be done in time $O(\mu(n_v) \cdot n_L \cdot |B|^{n_v})$ and the resulting set $\text{tfacts}(N,\beta)$ has worst-case size $O(n_A \cdot |B|^{n_v})$, where n_L is the number of literals in N, n_v is the maximum number of variables $|\text{vars}(A)|$ in any theory atom A in N, n_A is the number of different theory atoms in N, and $\mu(x)$ is the time needed to simplify a theory atom over x variables to a variable bound. The last set is denoted by $\text{sfacts}(N,\beta)$ and contains a fact $Q_{(P,i,S)}(a)$ for every fresh sort predicate $Q_{(P,i,S)}$ added during abstraction and every $a \in \beta(P,i) \cap S^{\mathcal{A}}$. This is enough to ensure that $Q_{(P,i,S)}$ evaluates to true for every test point assigned to the argument position (P,i) filtered by the sort S. Please note that already satifiability testing for BS clause sets is NEXPTIME-complete in general, and DEXPTIME-complete for the Horn case [26,33]. So when abstracting to a polynomially decidable clause set (ground HBS) an exponential factor is unavoidable.

Lemma 12. *N is equisatisfiable to its hammered version* $\text{tren}_N(N) \cup \text{tfacts}(N,\beta) \cup \text{sfacts}(N,\beta)$. *The conjecture $N \models \exists \bar{y}.Q(\bar{y})$ is false iff $N_D = \text{tren}'_N(N') \cup \text{tfacts}(N',\beta) \cup \text{sfacts}(N',\beta)$ is satisfiable with $N' = N \cup \{Q(\bar{y}) \rightarrow \bot\}$. The conjecture $N \models \forall \bar{y}.Q(\bar{y})$ is false iff $N_D = \text{tren}_N(N) \cup \text{tfacts}(N,\beta) \cup \text{sfacts}(N,\beta) \cup N_C$ is satisfiable.*

Note that $\text{tren}_N(N) \cup \text{tfacts}(N,\beta) \cup \text{sfacts}(N,\beta) \cup N_C$ is only a HBS clause set over a finite set of constants and not yet a Datalog input file. It is well known that such a formula can be transformed easily into a Datalog problem by adding a nullary predicate Goal and adding it as a positive literal to any clause without a positive literal. Querying for the Goal atom returns true if the HBS clause set was unsatisfiable and false otherwise.

Example 13. The hammered formula for example 6 looks as follows. The set of renamed clauses $\text{tren}_N(N)$ consists of all the previous clauses in N, except that inequalities have been abstracted to new first-order predicates:
$D'_1 : \text{SpeedTable}(0,2000,1350)$, $D'_2 : \text{SpeedTable}(2000,4000,1600)$,
$D'_3 : \text{SpeedTable}(4000,6000,1850)$, $D'_4 : \text{SpeedTable}(6000,8000,2100)$,
$C'_1 : P_{0 \leq x_p}(x_p), P_{x_p < 8000}(x_p) \rightarrow \text{Speed}(x_p)$,
$C'_2 : P_{x_1 \leq x_p}(x_1,x_p), P_{x_p < x_2}(x_p,x_2), \text{Speed}(x_p), \text{SpeedTable}(x_1,x_2,y) \rightarrow \text{IgnDeg}(x_p,y)$,
$C'_3 : \text{IgnDeg}(x_p,z) \rightarrow \text{ResArgs}(x_p)$, $C'_4 : \text{ResArgs}(x_p) \rightarrow \text{Conj}(x_p)$,
$C'_5 : P_{x_p \geq 8000}(x_p) \rightarrow \text{Conj}(x_p)$, $C'_6 : P_{x_p < 0}(x_p) \rightarrow \text{Conj}(x_p)$,
The set $\text{tfacts}(N,\beta)$ defines for which test points those new predicates evaluate to true: $\{P_{0 \leq x_p}(0)$, $P_{0 \leq x_p}(2000)$, $P_{0 \leq x_p}(4000)$, $P_{0 \leq x_p}(6000)$, $P_{0 \leq x_p}(8000)$, $P_{x_p < 8000}(-1)$, $P_{x_p < 8000}(0)$, $P_{x_p < 8000}(2000)$, $P_{x_p < 8000}(4000)$, $P_{x_p < 8000}(6000)$, $P_{x_1 \leq x_p}(0,0)$, $P_{x_1 \leq x_p}(0,2000)$, $P_{x_1 \leq x_p}(0,4000)$, $P_{x_1 \leq x_p}(0,6000)$, $P_{x_1 \leq x_p}(0,8000)$, $P_{x_1 \leq x_p}(2000,2000)$, $P_{x_1 \leq x_p}(2000,4000)$, $P_{x_1 \leq x_p}(2000,6000)$, $P_{x_1 \leq x_p}(2000,8000)$, $P_{x_1 \leq x_p}(4000,4000)$, $P_{x_1 \leq x_p}(4000,6000)$, $P_{x_1 \leq x_p}(4000,8000)$, $P_{x_1 \leq x_p}(6000,6000)$, $P_{x_1 \leq x_p}(6000,8000)$, $P_{x_p < x_2}(-1,2000)$, $P_{x_p < x_2}(0,2000)$, $P_{x_p < x_2}(-1,4000)$, $P_{x_p < x_2}(0,4000)$, $P_{x_p < x_2}(2000,4000)$, $P_{x_p < x_2}(-1,6000)$, $P_{x_p < x_2}(0,6000)$, $P_{x_p < x_2}(2000,6000)$,

Problem	Q	Status	$\lvert N \rvert$	vars	$\lvert B^m \rvert$	$\lvert\Delta_\phi\rvert$	SSPL	$\lvert B^s \rvert$	$\lvert\Delta^o_\phi\rvert$	SSPL06	vampire	spacer	z3	cvc4
lc_e1	∃	true	139	9	9	0	**< 0.1s**	45	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	0,1	**< 0.1s**
lc_e2	∃	false	144	9	9	0	**< 0.1s**	41	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	-	-
lc_e3	∃	false	138	9	9	0	**< 0.1s**	37	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	-	-
lc_e4	∃	true	137	9	9	0	**< 0.1s**	49	0	**< 0.1s**	**< 0.1s**	**< 0.1s**	**< 0.1s**	**< 0.1s**
lc_e5	∃	false	152	13	9	0	33.5s	-	-	N/A	**< 0.1s**	-	-	-
lc_e6	∃	true	141	13	9	0	42.8s	-	-	N/A	**0.1s**	3.3s	11.5s	0.4s
lc_e7	∃	false	141	13	9	0	41.4s	-	-	N/A	**< 0.1s**	7.6s	-	-
lc_e8	∃	false	141	13	9	0	32.5s	-	-	N/A	**< 0.1s**	2.1s	-	-
lc_u1	∀	false	139	9	9	27	**< 0.1s**	45	27	**< 0.1s**	**< 0.1s**	N/A	-	-
lc_u2	∀	false	144	9	9	27	**< 0.1s**	41	27	**< 0.1s**	**< 0.1s**	N/A	-	-
lc_u3	∀	true	138	9	9	27	**< 0.1s**	37	27	**< 0.1s**	**< 0.1s**	N/A	**< 0.1s**	**< 0.1s**
lc_u4	∀	false	137	9	9	27	**< 0.1s**	49	27	**< 0.1s**	**< 0.1s**	N/A	-	-
lc_u5	∀	false	154	13	9	3888	32.4s	-	-	N/A	**0.1s**	N/A	-	-
lc_u6	∀	true	154	13	9	3888	32.5s	-	-	N/A	**2.3s**	N/A	-	-
lc_u7	∀	true	141	13	9	972	32.3s	-	-	N/A	**0.2s**	N/A	-	-
lc_u8	∀	false	141	13	9	1259712	**48.8s**	-	-	N/A	2351.4s	N/A	-	-
ecu_e1	∃	false	757	10	96	0	**< 0.1s**	624	0	1.3s	0.2s	0.1s	-	-
ecu_e2	∃	true	757	10	96	0	**< 0.1s**	624	0	1.3s	0.2s	0.1s	1.4s	0.4s
ecu_e3	∃	false	775	11	196	0	50.1s	660	0	41.5s	3.1s	**0.1s**	-	-
ecu_u1	∀	true	756	11	96	37	**0.1s**	620	306	1.1s	32.8s	N/A	197.5s	0.4s
ecu_u2	∀	false	756	11	96	38	**0.1s**	620	307	1.1s	32.8s	N/A	-	-
ecu_u3	∀	true	745	9	88	760	**< 0.1s**	576	11360	0.7s	1.2s	N/A	239.5s	0.1s
ecu_u4	∀	true	745	9	486	760	**< 0.1s**	2144	237096	15.9s	1.2s	N/A	196.0s	0.1s
ecu_u5	∀	true	767	10	96	3900	**0.1s**	628	415296	31.9s	-	N/A	-	-
ecu_u6	∀	false	755	10	95	3120	**< 0.1s**	616	363584	14.4s	597.8	N/A	-	-
ecu_u7	∀	false	774	11	196	8400	**48.9s**	656	2004708	-	-	N/A	-	-
ecu_u8	∀	true	774	11	196	8400	**48.7s**	656	2004708	-	-	N/A	-	-

Fig. 2. Benchmark results and statistics

$P_{x_p<x_2}(4000{,}6000)$, $P_{x_p<x_2}(-1{,}8000)$, $P_{x_p<x_2}(0{,}8000)$, $P_{x_p<x_2}(2000{,}8000)$,
$P_{x_p<x_2}(4000{,}8000)$, $P_{x_p<x_2}(6000{,}8000)$, $P_{x_p\geq 800}(8000)$, $P_{x_p<0}(-1)\}$
$\mathrm{sfacts}(N,\beta)=\emptyset$ because there are no fresh sort predicates. The hammered negated conjecture is $N_C := \mathrm{Conj}(-1), \mathrm{Conj}(0), \mathrm{Conj}(2000), \mathrm{Conj}(4000), \mathrm{Conj}(6000), \mathrm{Conj}(8000) \rightarrow \bot$ and lets us derive false if and only if we can derive $\mathrm{Conj}(a)$ for all test points $a \in \beta(\mathrm{Conj},1)$.

4 Implementation and Experiments

We have implemented the sorted Datalog hammer as an extension to the SPASS-SPL system [11] (option -d) (SSPL in the table). By default the resulting formula is then solved with the Datalog reasoner VLog. The previously file-based combination with the Datalog reasoner VLog has been replaced by an integration of VLog into SPASS-SPL via the VLog API. We focus here only on the sorted extension and refer to [11] for an introduction into coupling of the two reasoners. Note that the sorted Datalog hammer itself is not fine tuned towards the capabilities of a specific Datalog reasoner nor VLog towards the sorted Datalog hammer.

In order to test the progress in efficiency of our sorted hammer, we ran the benchmarks of the lane change assistant and engine ECU from [11] plus more sophisticated, extended formalizations. While for the ECU benchmarks in [11] we modeled ignition timing computation adjusted by inlet temperature measurements, the new benchmarks take also gear box protection mechanisms into account. The lane change examples in [11] only simulated the supervisor for lane change assistants over some real-world instances. The new lane change benchmarks check properties for all potential inputs. The universal ones check that any suggested action by a lane change assistant is either proven as correct or disproven by our supervisor. The existential ones check safety properties, e.g., that the supervisor never returns both a proof and a disproof for the same input. We actually used SPASS-SPL to debug a prototype supervisor for lane change assistants during its development. The new lane change examples are based on versions generated during this debugging process where SPASS-SPL found the following bugs: (i) it did not always return a result, (ii) it declared actions as both safe and unsafe at the same time, and (iii) it declared actions as safe although they would lead to collisions. The supervisor is now fully verified.

The names of the problems are formatted so the lane change examples start with lc and the ECU examples start with ecu. Our benchmarks are prototypical for the complexity of HBS(SLA) reasoning in that they cover all abstract relationships between conjectures and HBS(SLA) clause sets. With respect to our two case studies we have many more examples showing respective characteristics. We would have liked to run benchmarks from other sources, but could not find any problems in the SMT-LIB [5,35] or CHC-COMP [2] benchmarks within the range of what our hammer can currently accept. Either the arithmetic part goes beyond SLA or there are further theories involved such as equality on first-order symbols.

For comparison, we also tested several state-of-the-art theorem provers for related logics (with the best settings we found): SPASS-SPL-v0.6 (SSPL06 in the table) that uses the original version of our Datalog Hammer [11] with settings `-d` for existential and `-d -n` for universal conjectures; the satisfiability modulo theories (SMT) solver *cvc4-1.8* [4] with settings `--multi-trigger-cache` `--full-saturate-quant`; the SMT solver *z3-4.8.12* [28] with its default settings; the constrained horn clause (CHC) solver *spacer* [24] with its default settings; and the first-order theorem prover *vampire-4.5.1* [37] with settings `--memory_limit 8000` `-p off`, i.e., with memory extended to 8GB and without proof output. For the SMT/CHC solvers, we directly transformed the benchmarks into their respective formats. Vampire gets the same input as VLog transformed into the TPTP format [39]. Our experiments with vampire investigate how superposition reasoners perform on the hammered benchmarks compared to Datalog reasoners.

For the experiments, we used the TACAS 22 artifact evaluation VM (Ubuntu 20.04 with 8 GB RAM and a single processor core) on a system with an Intel Core i7-9700K CPU with eight 3.60GHz cores. Each tool got a time limit of 40 minutes for each problem.

The table in Fig. 2 lists for each benchmark problem: the name of the problem (Problem); the type of conjecture (Q), i.e., whether the conjecture is existential ∃ or universal ∀; the status of the conjecture (Status); number of clauses ($|N|$); maximum number of variables in a clause (vars); the size of the largest test-point set introduced by the sorted/original Hammer (B^s/B^o); the size of the hammered universal conjecture ($|\Delta_\phi|/|\Delta_\phi^o|$ for sorted/original); the remaining columns list the time needed by the tools to solve the benchmark problems. An entry "N/A" means that the benchmark example cannot be expressed in the tools input format, e.g., it is not possible to encode a universal conjecture (or, to be more precise, its negation) in the CHC format and SPASS-SPL-v0.6 is not sound when the problem contains integer variables. An entry "-" means that the tool ran out of time, ran out of memory, exited with an error or returned unknown.

The experiments show that SPASS-SPL (with the sorted Hammer) is orders of magnitudes faster than SPASS-SPL-v0.6 (with the original Hammer) on problems with universal conjectures. On problems with existential conjectures, we cannot observe any major performance gain compared to the original Hammer. Sometimes SPASS-SPL-v0.6 is even slightly faster (e.g. ecu_e3). Potential explanations are: First, the number of test points has a much larger impact on universal conjectures because the size of the hammered universal conjecture increases exponentially with the number of test points. Second, our sorted Hammer needs to generate more abstracted theory facts than the original Hammer because the latter can reuse abstraction predicates for theory atoms that are identical upto variable renaming. The sorted Hammer can reuse the same predicate only if variables also range over the same sets of test points, which we have not yet implemented.

Compared to the other tools, SPASS-SPL is the only one that solves all problems in reasonable time. It is also the only solver that can decide in reasonable time whether a universal conjecture is *not* a consequence. This is not surprising because to our knowledge SPASS-SPL is the only theorem prover that implements a decision procedure for HBS(SLA). On the problems with existential conjectures, our tool-chain solves all of the problems in under a minute and with comparable times to the best tool for the problem. The only exception are problems that contain a lot of superfluous clauses, i.e., clauses that are not needed to confirm/refute the conjecture. The reason might be that VLog derives all facts for the input problem in a breadth-first way, which is not very efficient if there are a lot of superfluous clauses. Vampire coupled with our sorted Hammer returns the best results for those problems. Vampire performed best on the hammered problems among all first-order theorem provers we tested, including iProver [25], E [38], and SPASS [40]. We tested all provers in default theorem proving mode with adjusted memory limits. The experiments with the first-order provers showed that our hammer also works reasonably well for them, but they do not scale well if the size and the complexity of the universal conjectures increases. For problems with existential conjectures, the CHC solver spacer is often the best, but as a trade-off it is unable to handle universal conjectures. The instantiation techniques employed by cvc4 are good for proving some universal conjectures, but both SMT solvers seem to be unable to disprove conjectures.

5 Conclusion

We have presented an extension of our previous Datalog hammer [11] supporting a more expressive input logic resulting in more elegant and more detailed supervisor formalizations, and through a soft typing discipline supporting more efficient reasoning. Our experiments show, compared to [11], that our performance on existential conjectures is at the same level as SMT and CHC solvers. The complexity of queries we can handle in reasonable time has significantly increased, see Section 4, Figure 2. Still SPASS-SPL is the only solver that can prove and disprove universal queries. The file interface between SPASS-SPL and VLog has been replaced by a close coupling resulting in a more comfortable application.

Our contribution here solves the third point for future work mentioned in [11] although there is still room to also improve our soft typing discipline. In the future, we want SPASS-SPL to produce explications that prove that its translations are correct. Another direction is to exploit specialized Datalog expressions and techniques, e.g., aggregation and stratified negation, to increase the efficiency of our tool-chain and to lift some restrictions from our input formulas. Finally, our hammer can be seen as part of an overall reasoning methodology for the class of BS(LA) formulas which we presented in [12]. We will implement and further develop this methodology and integrate our Datalog hammer.

Acknowledgments: This work was funded by DFG grant 389792660 as part of TRR 248 (CPEC), by BMBF in project ScaDS.AI, and by the Center for Advancing Electronics Dresden (cfaed). We thank our anonymous reviewers for their constructive comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Angelis, E.D., K, H.G.V.: Constrained horn clauses (chc) competition (2022), https://chc-comp.github.io/
3. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. Applicable Algebra in Engineering, Communication and Computing, AAECC **5**(3/4), 193–212 (1994)
4. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, LNCS, vol. 6806 (2011)
5. Barrett, C.W., de Moura, L.M., Ranise, S., Stump, A., Tinelli, C.: The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In: Barner, S., Harris, I.G., Kroening, D., Raz, O. (eds.) Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6504, p. 3. Springer (2010)
6. Baumgartner, P., Waldmann, U.: Hierarchic superposition revisited. In: Lutz, C., Sattler, U., Tinelli, C., Turhan, A., Wolter, F. (eds.) Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 11560, pp. 15–56. Springer (2019)
7. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015)
8. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6173, pp. 107–121. Springer (2010)
9. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., González, L., Krötzsch, M., Marx, M., Murali, H.K., Weidenbach, C.: Artifact for a sorted Datalog hammer for supervisor verification conditions modulo simple linear arithmetic (Jan 2022). https://doi.org/10.5281/zenodo.5888272
10. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., González, L., Krötzsch, M., Marx, M., Murali, H.K., Weidenbach, C.: A sorted Datalog hammer for supervisor verification conditions modulo simple linear arithmetic. CoRR **abs/2201.09769** (2022), https://arxiv.org/abs/2201.09769
11. Bromberger, M., Dragoste, I., Faqeh, R., Fetzer, C., Krötzsch, M., Weidenbach, C.: A Datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In: Reger, G., Konev, B. (eds.) Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, United Kongdom, September 8-10, 2021. Proceedings. Lecture Notes in Computer Science, vol. 12941, pp. 3–24. Springer (2021)
12. Bromberger, M., Fiori, A., Weidenbach, C.: Deciding the bernays-schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 511–533. Springer (2021)
13. Carral, D., Dragoste, I., González, L., Jacobs, C., Krötzsch, M., Urbani, J.: VLog: A rule engine for knowledge graphs. In: Ghidini et al., C. (ed.) Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II). LNCS, vol. 11779, pp. 19–35. Springer (2019)
14. Cimatti, A., Griggio, A., Redondi, G.: Universal invariant checking of parametric systems with quantifier-free SMT reasoning. In: Proc. CADE-28 (2021), to appear
15. Downey, P.J.: Undecidability of presburger arithmetic with a single monadic predicate letter. Tech. rep., Center for Research in Computer Technology, Harvard University (1972)

16. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22**(3), 364–418 (1997)
17. Faqeh, R., Fetzer, C., Hermanns, H., Hoffmann, J., Klauck, M., Köhl, M.A., Steinmetz, M., Weidenbach, C.: Towards dynamic dependable systems through evidence-based continuous certification. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12477, pp. 416–439. Springer (2020)
18. Fiori, A., Weidenbach, C.: SCL with theory constraints. CoRR **abs/2003.04627** (2020), https://arxiv.org/abs/2003.04627
19. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer (2009)
20. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012)
21. Hillenbrand, T., Weidenbach, C.: Superposition for bounded domains. In: Bonacina, M.P., Stickel, M. (eds.) McCune Festschrift. LNCS, vol. 7788, pp. 68–100. Springer (2013)
22. Horbach, M., Voigt, M., Weidenbach, C.: On the combination of the bernays-schönfinkel-ramsey fragment with simple linear integer arithmetic. 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. 77–94. Springer (2017)
23. Horbach, M., Voigt, M., Weidenbach, C.: The universal fragment of presburger arithmetic with unary uninterpreted predicates is undecidable. CoRR **abs/1703.01212** (2017)
24. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 17–34. Springer (2014)
25. Korovin, K.: iprover - an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5195, pp. 292–298. Springer (2008)
26. Lewis, H.R.: Complexity results for classes of quantificational formulas. Journal of Compututer and System Sciences **21**(3), 317–353 (1980)
27. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal **36**(5), 450–462 (1993)
28. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 4963 (2008)
29. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
30. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). Journal of the ACM **53**, 937–977 (November 2006)
31. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
32. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Handbook of Automated Reasoning, pp. 335–367. Elsevier and MIT Press (2001)
33. Plaisted, D.A.: Complete problems in the first-order predicate calculus. Journal of Computer and System Sciences **29**, 8–35 (1984)
34. Ranise, S.: On the verification of security-aware e-services. Journal of Symbolic Compututation **47**(9), 1066–1088 (2012)

35. Ranise, S., Tinelli, C., Barrett, C., Fontaine, P., Stump, A.: Smt-lib the satisfiability modulo theories library (2022), https://smtlib.cs.uiowa.edu/
36. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 112–131. Springer (2018)
37. Riazanov, A., Voronkov, A.: The design and implementation of vampire. AI Communications **15**(2-3), 91–110 (2002)
38. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. pp. 495–507. No. 11716 in LNAI, Springer (2019)
39. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017)
40. Weidenbach, C., Dimova, D., Fietzke, A., Suda, M., Wischnewski, P.: Spass version 3.5. In: Schmidt, R.A. (ed.) 22nd International Conference on Automated Deduction (CADE-22). Lecture Notes in Artificial Intelligence, vol. 5663, pp. 140–145. Springer, Montreal, Canada (August 2009)

Model Checking and Verification

Property Directed Reachability for Generalized Petri Nets

Nicolas Amat[1](✉), Silvano Dal Zilio[1], and Thomas Hujsa[1]

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
namat@laas.fr

Abstract. We propose a semi-decision procedure for checking generalized reachability properties, on generalized Petri nets, that is based on the Property Directed Reachability (PDR) method. We actually define three different versions, that vary depending on the method used for abstracting possible witnesses, and that are able to handle problems of increasing difficulty. We have implemented our methods in a model-checker called SMPT and give empirical evidences that our approach can handle problems that are difficult or impossible to check with current state of the art tools.

Keywords: Petri nets · Model Checking · Reachability · SMT solving

1 Introduction

We propose a new semi-decision procedure for checking reachability properties on generalized Petri nets, meaning that we impose no constraints on the weights of the arcs and do not require a finite state space. We also consider a generalized notion of reachability, in the sense that we can not only check the reachability of a given state, but also if it is possible to reach a marking that satisfies a combination of linear constraints between places, such as $(p_0 + p_1 = p_2 + 2) \wedge (p_1 \leqslant p_2)$ for example. Another interesting feature of our approach is that we are able to return a "certificate of invariance", in the form of an *inductive linear invariant* [26], when we find that a constraint is true on all the reachable markings. To the best of our knowledge, there is no other tool able to compute such certificates in the general case.

Our approach is based on an extension of the Property Directed Reachability (PDR) method, originally developed for hardware model-checking [8,9], to the case of Petri nets. We actually define three variants of our algorithm—two of them completely new when compared to our previous work [1]—that vary based on the method used for generalizing possible witnesses and can handle problems of increasing difficulty.

Reachability for Petri nets is an important and difficult problem with many practical applications: obviously for the formal verification of concurrent systems, but also for the study of diverse types of protocols (such as biological or business processes); the verification of software systems; the analysis of infinite

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 505–523, 2022.
https://doi.org/10.1007/978-3-030-99524-9_28

state systems; etc. It is also a timely subject, as shown by recent publications on this subject [7,15], but also with the recent progress made on settling its theoretical complexity [12,13], which asserts that reachability is Ackermann-complete, and therefore inherently more complex than, say, the coverability problem. A practical consequence of this "inherent complexity", and a general consensus, is that we should not expect to find a one-size-fits-all algorithm that could be usable in practice. A better strategy is to try to improve the performances on some cases—for example by developing new tools, or optimizations, that may perform better on some examples—or try to improve "expressiveness"—by finding algorithms that can manage new cases, that no other tool can handle.

This wisdom is illustrated by the current state of the art at the Model Checking Contest (MCC) [3], a competition of model-checkers for Petri nets that includes an examination for the reachability problem. Albeit strongly oriented towards the analysis of bounded nets. As a matter of fact, the top three tools in recent competitions—ITS-Tools [30], LoLA [34], and Tapaal [14]—all rely on a portfolio approach. Methods that have been proposed in this context include the use of symbolic techniques, such as k-induction [31]; abstraction refinement [10]; the use of standard optimizations with Petri nets, like stubborn sets or structural reductions; the use of the "state equation"; reduction to integer linear programming problems; etc.

The results obtained during the MCC highlight the very good performances achieved when putting all these techniques together, on bounded nets, with a collection of randomly generated properties. Another interesting feedback from the MCC is that simulation techniques are very good at finding a *counter-example* when a property is not an invariant [7,31].

In our work, we seek improvements in terms of both *performance* and *expressiveness*. We also target what we consider to be a difficult, and less studied area of research: procedures that can be applied when a property is an invariant and when the net is unbounded, or its state space cannot be fully explored. We also focus on the verification of "genuine" reachability constraints, which are not instances of a coverability problem. These properties are seldom studied in the context of unbounded nets. Interestingly enough, our work provides a simple explanation of why coverability problems are also "simpler" in the case of PDR; what we will associate with the notion of *monotonic formulas*.

Concerning performances, we propose a method based on a well-tried symbolic technique, PDR, that has proved successful with unbounded model-checking and when used together with SMT solvers [11,22]. Concerning expressiveness, we define a small benchmark of "difficult nets": a set of synthetic examples, representative of patterns that can make the reachability problem harder.

Outline and Contributions. We define background material on Petri nets in Sect. 2, where we use Linear Integer Arithmetic (LIA) formulas to reason about nets. Section 3 describes our decision method, based on PDR and SMT solvers, for checking the satisfiability of linear invariants over the reachable states of a Petri net. Our method builds sequences of incremental invariants using both a property that we want to disprove, and a stepwise approximation of the reachability relation. It also relies on a generalization step where we can abstract possible "bad states" into clauses that are propagated in order to find a counter-example, or to block inconsistent states.

We describe a first generalization method, based on the upset of markings, that is able to deal with coverability properties. We propose a new, dual variant based on the concept of *hurdles* [21], that is without restrictions on the properties. In this method, the goal is to block bad sequences of transitions instead of bad states. We show how this approach can be further improved by defining a notion of saturated transition sequence, at the cost of adding universal quantification in our SMT problems.

We have implemented our approach in an open-source tool, called SMPT, and compare it with other existing tools. In this context, one of our contributions is the definition of a set of difficult nets, that characterizes classes of difficult reachability problems.

2 Petri Nets and Linear Reachability Constraints

Let $\mathbb{N}$ denote the set of natural numbers and $\mathbb{Z}$ the set of integers. Assuming P is a finite, totally ordered set $\{p_1, \ldots, p_n\}$, we denote by $\mathbb{N}^P$ the set of mappings from $P \to \mathbb{N}$ and we overload the addition, subtraction and comparison operators $(=, \geq, \leq)$ to act as their component-wise equivalent on mappings. A QF-LIA formula F, with support in P, is a Boolean combination of atomic propositions of the form $\alpha \sim \beta$, where $\sim$ is one of $=, \leq$ or $\geq$ and α, β are *linear expressions*, that is, linear combinations of elements in $\mathbb{N} \cup P$. We simply use the term *linear constraint* to describe F.

A *Petri net* N is a tuple $(P, T, \mathbf{pre}, \mathbf{post})$ where $P = \{p_1, \ldots, p_n\}$ is a finite set of places, T is a finite set of transitions (disjoint from P), and $\mathbf{pre} : T \to \mathbb{N}^P$ and $\mathbf{post} : T \to \mathbb{N}^P$ are the pre- and post-condition functions (also called the flow functions of N). A state m of a net, also called a *marking*, is a mapping of $\mathbb{N}^P$. We say that the marking m assigns $m(p_i)$ tokens to place p_i. A marked net (N, m_0) is a pair composed from a net and an initial marking m_0.

A transition $t \in T$ is *enabled* at marking $m \in \mathbb{N}^P$ when $m \geqslant \mathbf{pre}(t)$. When t is enabled at m, we can fire it and reach another marking $m' \in \mathbb{N}^P$ such that $m' = m - \mathbf{pre}(t) + \mathbf{post}(t)$. We denote this transition $m \xrightarrow{t} m'$. The difference between m and m' is a mapping $\Delta(t) = \mathbf{post}(t) - \mathbf{pre}(t)$ in $\mathbb{Z}^P$, also called the *displacement* of t.

By extension, we say that a *firing sequence* $\sigma = t_1 \ldots t_k \in T^*$ can be fired from m, denoted $m \overset{\sigma}{\Rightarrow} m'$, if there exist markings $m_0, \ldots, m_k$ such that $m = m_0$, $m' = m_k$ and $m_i \xrightarrow{t_{i+1}} m_{i+1}$ for all $i < k$. We can also simply write $m \to^\star m'$. In this case, the displacement of σ is the mapping $\Delta(\sigma) = \Delta(t_1) + \cdots + \Delta(t_k)$. We denote by $R(N, m_0)$ the set of markings reachable from m_0 in N. A marking m is k-bounded when each place has at most k tokens. By extension, we say that a marked Petri net (N, m_0) is bounded when there is k such that all reachable markings are k-bounded.

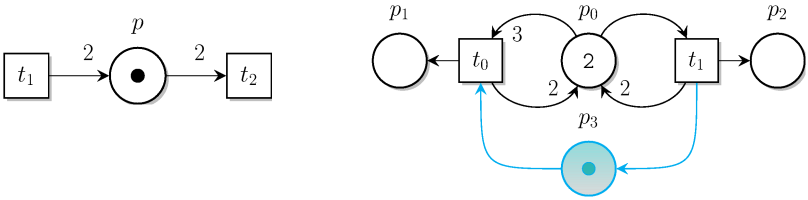


Fig. 1. Two examples of Petri nets: Parity (left) and PGCD (right).

While reachable states are computed by adding a linear combination of "displacements" (vectors in $\mathbb{Z}^P$), the set $R(N, m_0)$ is not necessarily semilinear or, equivalently, definable using Presburger arithmetic [20,26]. This is a consequence of the constraint that transitions must be enabled before firing. But there is still some structure to the set $R(N, m_0)$, like for instance the following monotonicity constraint:

$$\forall m \in \mathbb{N}^P \,.\, m_1 \overset{\sigma}{\Longrightarrow} m_2 \;\text{ implies }\; m_1 + m \overset{\sigma}{\Longrightarrow} m_2 + m \tag{H1}$$

We have other such results, such as with the notion of *hurdle* [21]. Just as $\mathbf{pre}(t)$ is the smallest marking for which a given transition t is enabled, there is a smallest marking at which a given firing sequence σ is fireable. This marking, denoted by $H(\sigma)$, has a simple inductive definition:

$$H(t) = \mathbf{pre}(t) \quad \text{and} \quad H(\sigma_1 \cdot \sigma_2) = \max\left(H(\sigma_1), H(\sigma_2) - \Delta(\sigma_1)\right) \tag{H2}$$

Given this notion of hurdles, we obtain that $m \overset{\sigma}{\Longrightarrow} m'$ if and only if (1) the sequence σ is enabled: $m \geqslant H(\sigma)$, and (2) $m' = m + \Delta(\sigma)$. We use this result in the second variant of our method.

We can go a step further and characterize a necessary and sufficient condition for firing the sequence $\sigma.\sigma^k$, meaning firing the same sequence more than once. Given $\Delta(\sigma)$, a place p with a negative displacement (say $-d$) means that we "loose" d token each time we fire σ. Hence we should budget d tokens in p for each new iteration. Therefore we have $m \overset{\sigma}{\Longrightarrow} \overset{\sigma^k}{\Longrightarrow} m'$ if and only if (1) $m \geqslant H(\sigma) + k \cdot \max(\mathbf{0}, -\Delta(\sigma))$, and (2) $m' = m + (k+1) \cdot \Delta(\sigma)$. Equivalently, if we denote by m^+ the "positive" part of mapping m, such that $m^+(p) = 0$ when $m(p) \leqslant 0$ and $m^+(p) = m(p)$ otherwise, we have:

$$H(\sigma^{k+1}) = \max\left(H(\sigma), H(\sigma) - k \cdot \Delta(\sigma)\right) = H(\sigma) + k \cdot (-\Delta(\sigma))^+ \tag{H3}$$

Examples. We give two simple examples of unbounded nets in Fig. 1, which are both part of our benchmark. Parity has a single place, hence its state space can be interpreted as a subset of $\mathbb{N}$: with an initial marking of 1, this is exactly the set of odd numbers (and therefore state 0 is not reachable). We are in a special case where the set $R(N, m_0)$ is semilinear. For instance, it can be seen as solution to the constraint $\exists k.(p = 2k + 1)$, or equivalently $p \equiv 1 \pmod 2$. But it cannot be expressed with a linear constraint involving only the variable

p without quantification or modulo arithmetic. This example can be handled by most of the tools used in our experiments, e.g. with the help of k-induction.

In PGCD, transitions t_0/t_1 can decrement/increment the marking of p_0 by 1. Nonetheless, with this initial state, it is the case that the number of occurrences of t_0 is always less than the one of t_1 in any feasible sequence σ. Hence the two predicates $p_0 \geq 2$ and $p_2 \geq p_1$ are valid invariants. (Since some tools do not accept literals of the form $p \geq q$, we added the "redundant" place p_3 so we can restate our second invariant as $p_3 \geq 1$.) These invariants cannot be proved by reasoning only on the displacements of traces (using the state equation) and are already out of reach for LoLA or Tapaal.

Linear Reachability Formulas. We can revisit the semantics of Petri nets using linear predicates. In the following, we use $\boldsymbol{p}$ for the vector $(p_1, \ldots, p_n)$, and $F(\boldsymbol{p})$ for a formula with variables in P. We also simply use $F(\boldsymbol{\alpha})$ for the substitution $F\{p_1 \leftarrow \alpha_1\} \ldots \{p_n \leftarrow \alpha_n\}$, with $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ a sequence of linear expressions. We say that a mapping m of $\mathbb{N}^P$ is a *model* of F, denoted $m \models F$, if the ground formula $F(m) = F(m(p_1), \ldots, m(p_n))$ is true. Hence we can also interpret F as a predicate over markings. Finally, we define the semantics of F as the set $[\![F]\!] = \{m \in \mathbb{N}^P \mid m \models F\}$.

As usual, we say that a predicate F is *valid*, denoted $\models F$, when all its interpretations are true ($[\![F]\!] = \mathbb{N}^P$); and that F is *unsatisfiable* (or simply `unsat`), denoted $\not\models F$, when $[\![F]\!] = \emptyset$.

We can define many properties on the markings of a net N using this framework. For instance, we can model the set of markings m such that some transition t is enabled using predicate ENBL_t (see Equation (2) below). We can also define a linear predicate to describe the relation between the markings before and after some transition t fires. To this end, we use a vector $\boldsymbol{p'}$ of "primed variables" $(p'_1, \ldots, p'_n)$, where p'_i will stand for the marking of place p_i after a transition is fired. With this convention, formula $\mathrm{FIRE}_t(\boldsymbol{p}, \boldsymbol{p'})$ is such that $\mathrm{FIRE}_t(m, m')$ entails $m \xrightarrow{t} m'$ or $m = m'$ when t is enabled at m. With all these notations, we can define a predicate $\mathrm{T}(\boldsymbol{p}, \boldsymbol{p'})$ that "encodes" the effect of firing at most one transition in the net N.

$$\mathrm{GEQ}_m(\boldsymbol{p}) \overset{\text{def}}{=} \textstyle\bigwedge_{i \in 1..n} (p_i \geqslant m(p_i)) \tag{1}$$

$$\mathrm{ENBL}_t(\boldsymbol{p}) \overset{\text{def}}{=} \textstyle\bigwedge_{i \in 1..n} (p_i \geqslant \mathbf{pre}(t)(p_i)) \;=\; \mathrm{GEQ}_{H(t)}(\boldsymbol{p}) \tag{2}$$

$$\mathit{\Delta}_t(\boldsymbol{p}, \boldsymbol{p'}) \overset{\text{def}}{=} \textstyle\bigwedge_{i \in 1..n} (p'_i = p_i + \mathbf{post}(t)(p_i) - \mathbf{pre}(t)(p_i)) \tag{3}$$

$$\mathrm{EQ}(\boldsymbol{p}, \boldsymbol{p'}) \overset{\text{def}}{=} \textstyle\bigwedge_{i \in 1..n} (p'_i = p_i) \tag{4}$$

$$\mathrm{FIRE}_t(\boldsymbol{p}, \boldsymbol{p'}) \overset{\text{def}}{=} \mathrm{EQ}(\boldsymbol{p}, \boldsymbol{p'}) \vee (\mathrm{ENBL}_t(\boldsymbol{p}) \wedge \mathit{\Delta}_t(\boldsymbol{p}, \boldsymbol{p'})) \tag{5}$$

$$\mathrm{T}(\boldsymbol{p}, \boldsymbol{p'}) \overset{\text{def}}{=} \mathrm{EQ}(\boldsymbol{p}, \boldsymbol{p'}) \vee \textstyle\bigvee_{t \in T} (\mathrm{ENBL}_t(\boldsymbol{p}) \wedge \mathit{\Delta}_t(\boldsymbol{p}, \boldsymbol{p'})) \tag{6}$$

In our work, we focus on the verification of *safety* properties on the reachable markings of a marked net (N, m_0). Examples of properties that we want to check include: checking if some transition t is enabled (commonly known as *quasi-liveness*); checking if there is a deadlock; checking whether some linear invariant between place markings is true; ... All properties that can be expressed using a linear predicate.

Definition 1 (Linear Invariants and Inductive Predicates). *A linear predicate F is an invariant on (N, m_0) if and only if we have $m \models F$ for all $m \in R(N, m_0)$. It is inductive if for all markings m we have $m \models F$ and $m \to m'$ entails $m' \models F$.*

It is possible to characterize inductive predicates using our logical framework. Indeed, F is inductive if and only if the QF-LIA formula (i) $F(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \neg F(\boldsymbol{p'})$ is `unsat`. Also, an inductive formula is an invariant when (ii) $m_0 \models F$, or equivalently $\models F(m_0)$. As a consequence, a sufficient condition for a predicate F to be invariant is to have both conditions (i) and (ii); conditions that can be checked using a SMT solver. Unfortunately, the predicates that we need to check are often not inductive. In this case, the next best thing is to try to build an inductive invariant, say R, such that $[\![R]\!] \subseteq [\![F]\!]$ (or equivalently $R \wedge \neg F$ `unsat`). This predicate provides a certificate of invariance that can be checked independently.

Lemma 1 (Certificate of Invariance). *A sufficient condition for F to be invariant on (N, m_0) is to exhibit a linear predicate R that is (i) initial: $R(m_0)$ **valid**; (ii) inductive: $R(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \neg R(\boldsymbol{p}\ )$ **unsat**; and (iii) that entails F, for instance: $R \wedge \neg F$ **unsat**.*

This result is in line with a property proved by Leroux [26], which states that when a final configuration m is not reachable there must exist a Presburger inductive invariant that contains m_0 but does not contain m. This result does not explain how to effectively compute such an invariant. Moreover, in our case, we provide a method that works with general linear predicates, and not only with single configurations. On the other side of the coin, given the known results about the complexity of the problem, we do not expect our procedure to be complete in the general case.

In the next section, we show how to (potentially) find such certificates using an adaptation of the PDR method. An essential component of PDR is to abstract a "scenario" leading to the model of some property F—say a transition $m \overset{\sigma}{\Rightarrow} m'$ with $m' \models F$—into a predicate that contains m (and potentially many more similar scenarios). More generally, a *generalization* of the trio (m, σ, F) is a predicate G satisfied by m such that $m_1 \models G$ entails that there is $m_1 \to^\star m_2$ with $m_2 \models F$.

We can use properties (H1)–(H3), defined earlier, to build generalizations.

Lemma 2 (Generalization). *Assume we have a scenario such that $m \overset{\sigma}{\Rightarrow} m'$ and $m' \models F$. We have three possible generalizations of the trio (m, σ, F).*

(G1) *If property F is monotonic, then $m_1 \models \mathrm{GEQ}_m(\boldsymbol{p})$ implies there is $m_2 \geqslant m'$ such that $m_1 \overset{\sigma}{\Rightarrow} m_2$ and $m_2 \models F$.*

(G2) *If* $m_1 \models \mathrm{GEQ}_{H(\sigma)}(\boldsymbol{p}) \wedge F(\boldsymbol{p} + \Delta(\sigma))$ *then* $m_1 \overset{\sigma}{\Rightarrow} m_2$ *and* $m_2 \models F$.

(G3) *Assume* a, b *are mappings of* $\mathbb{N}^P$ *such that* $a = H(\sigma)$ *and* $b = (-\Delta(\sigma))^+$, *with the notations used in (H3). Then*

$$m_1 \models \exists k. \left(\begin{matrix} [\bigwedge_{i \in 1..n}(p_i \geqslant a(i) + k \cdot b(i))] \\ \wedge F(\boldsymbol{p} + (k+1) \cdot \Delta(\sigma)) \end{matrix} \right) \quad \textit{implies} \quad \begin{cases} \exists k. m_1 \xRightarrow{\sigma^{k+1}} m_2 \\ \textit{and } m_2 \models F \end{cases}$$

Proof. Each property is a direct result of properties (H1) to (H3). □

Property (G3) is the first and only instance of linear formula using an extra variable, k, that is not in P. The result is still a linear formula though, since we never need to use the product of two variables. This generalization is used when we want to "saturate the sequence σ". This is the only situation where we may need to deal with quantified LIA formulas. Another solution would be to replace each quantification with the use of modulo arithmetic, but this operation may be costly and could greatly increase the size of our formulas. It would also not cut down the complexity of the SMT problems.

3 Property Directed Reachability

Some symbolic model-checking procedure, such as BMC [6] or k-induction [28], are a good fit when we try to find counter-examples on infinite-state systems. Unfortunately, they may perform poorly when we want to check an invariant. In this case, adaptations of the PDR method [8,9] (also known as IC3, for "Incremental Construction of Inductive Clauses for Indubitable Correctness") have proved successful.

We assume that we start with an initial state m_0 satisfying a linear property, $\mathbb{I}$, and that we want to prove that property $\mathbb{P}$ is an invariant of the marked net (N, m_0). (We use blackboard bold symbols to distinguish between parameters of the problem, and formulas that we build for solving it.) We define $\mathbb{F} = \neg\mathbb{P}$ as the "set of feared events"; such that $\mathbb{P}$ is not an invariant if we can find m in $R(N, m_0)$ such that $m \models \mathbb{F}$. To simplify the presentation, we assume that $\mathbb{F}$ is a conjunction of literals (a cube), meaning that $\mathbb{P}$ is a clause. In practice, we assume that $\mathbb{F}$ is in Disjunctive Normal Form.

PDR is a combination of induction, over-approximation, and SAT or SMT solving. The goal is to build an incremental sequence of predicates $F_0, \dots, F_k$ that are "inductive relative to stepwise approximations": such that $m \models F_i$ and $m \to m'$ entails $m' \models F_{i+1}$, but not $m' \models \mathbb{F}$. The method stops when it finds a counter-example, or when we find that one of the predicates F_i is inductive.

We adapt the PDR approach to Petri nets, using linear predicates and SMT solvers for the QF-LIA and LIA logics in order to learn, generalize, and propagate new clauses. The most innovative part of our approach is the use of specific "generalization algorithms" that take advantage of the Petri nets theory, like the use of hurdles for example. Our implementation follows closely the algorithm for IC3 described in [9] and, for the sake of brevity, we only give the pseudo-code for the four main functions.

```
Function prove($\mathbb{I}$, $\mathbb{F}$: linear predicates)
    Result: $\bot$ if $\mathbb{F}$ is reachable ($\mathbb{P} = \neg\mathbb{F}$ is not an invariant), otherwise $\top$
 1  if sat($\mathbb{I}(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \mathbb{F}(\boldsymbol{p'})$) then
 2      return $\bot$
 3  $k \leftarrow 1$, $F_0 \leftarrow \mathbb{I}$, $F_1 \leftarrow \mathbb{P}$
 4  while $\top$ do
 5      if not strengthen($k$) then
 6          return $\bot$
 7      propagateClauses($k$)
 8      if $\mathrm{CL}(F_i) = \mathrm{CL}(F_{i+1})$ for some $1 \leqslant i \leqslant k$ then
 9          return $\top$
10      $k \leftarrow k + 1$
```

The main function, prove, computes an *Over Approximated Reachability Sequence* (OARS) $(F_0, \ldots, F_k)$ of linear predicates, called *frames*, with variables in $\boldsymbol{p}$. An OARS meets the following constraints: (1) it is monotonic: $F_i \wedge \neg F_{i+1}$ unsat for $0 \leqslant i < k$; (2) it contains the initial states: $\mathbb{I} \wedge \neg F_0$ unsat; (3) it does not contain feared states: $F_i \wedge \mathbb{F}$ unsat for $0 \leqslant i \leqslant k$; and (4) it satisfies *consecution*: $F_i(\boldsymbol{p}) \wedge \mathrm{T}(\boldsymbol{p}, \boldsymbol{p'}) \wedge \neg F_{i+1}(\boldsymbol{p'})$ unsat for $0 \leqslant i < k$.

By construction, each frame F_i in the OARS is defined as a set of clauses, $\mathrm{CL}(F_i)$, meaning that F_i is built as a formula in CNF: $F_i = \bigwedge_{cl \in \mathrm{CL}(F_i)} cl$. We also enforce that $\mathrm{CL}(F_{i+1}) \subseteq \mathrm{CL}(F_i)$ for $0 \leqslant i < k$, which means that the monotonicity property between frames is trivially ensured.

The body of function prove contains a *main iteration* (line 4) that increases the value of k (the number of levels of the OARS). At each step, we enter a second, minor iteration (line 2 in function strengthen), where we generate new minimal inductive clauses that will be propagated to all the frames. Hence both the length of the OARS, and the set of clauses in its frames, increase during computation. The procedure stops when we find an index i such that $F_i = F_{i+1}$. In this case we know that F_i is an inductive invariant satisfying $\mathbb{P}$. We can also stop during the iteration if we find a counter-example (a model m of $\mathbb{F}$). In this case, we can also return a trace leading to m.

When we start the first minor iteration, we have $k = 1$, $F_0 = \mathbb{I}$ and $F_1 = \mathbb{P}$. If we have $F_k(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \mathbb{F}(\boldsymbol{p})$ unsat, it means that $\mathbb{P}$ is inductive, so we can stop and return that $\mathbb{P}$ is an invariant. Otherwise, we proceed with the strengthen phase, where each model of $F_k(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \mathbb{F}(\boldsymbol{p})$ becomes a potential counter-example, or *witness*, that we need to "block" (line 3–5 of function strengthen).

Instead of blocking only one witness, we first generalize it into a predicate that abstracts similar dangerous states (see the call to generalizeWitness). This is done by applying one of the three generalization results in Lemma 2. We give more details about this step later. By construction, each generalization is a cube s (a conjunction of literals). Hence, when we block it, we learn new clauses from $\neg s$ that can be propagated to the previous frames.

```
Function strengthen($k$ : current level)
1 try:
2     while $(m \xrightarrow{t} m') \models F_k(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \mathbb{F}(\boldsymbol{p'})$ do
3         $s \leftarrow$ generalizeWitness($m$, $t$, $\mathbb{F}$)
4         $n \leftarrow$ inductivelyGeneralize($s$, $k$ - $2$, $k$)
5         pushGeneralization($\{(s, n+1)\}$, $k$)
6     return $\top$
7 catch counter example:
8     return $\bot$
```

```
Function inductivelyGeneralize($s$ : cube, $min$: level, $k$: level)
1 if $min < 0$ and sat($F_0(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge s(\boldsymbol{p'})$) then
2     raise Counterexample
3 for $i \leftarrow \max(1, min + 1)$ to $k$ do
4     if sat($F_i(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge \neg s(\boldsymbol{p}) \wedge s(\boldsymbol{p'})$) then
5         generateClause($s$, $i$-$1$, $k$)
6         return $i - 1$
7 generateClause($s$, $k$, $k$)
8 return $k$
```

Before pushing a new clause, we test whether s is reachable from previous frames. We take advantage of this opportunity to find if we have a counter-example and, if not, to learn new clauses in the process. This is the role of functions pushGeneralization and inductivelyGeneralize.

We find a counter example (in the call to inductivelyGeneralize) if the generalization from a witness found at level k, say s, reaches level 0 and $F_0(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge s(\boldsymbol{p'})$ is satisfiable (line 1 in inductivelyGeneralize). Indeed, it means that we can build a trace from $\mathbb{I}$ to $\mathbb{F}$ by going through $F_1, \ldots, F_k$.

The method relies heavily on checking the satisfiability of linear formulas in QF-LIA, which is achieved with a call to a SMT solver. In each function call, we need to test if predicates of the form $F_i \wedge T \wedge G$ are unsat and, if not, enumerate its models. To accelerate the strengthening of frames, we also rely on the unsat core of properties in order to compute a *minimal inductive clause* (MIC).

Our approach is parametrized by a generalization function (generalizeWitness) that is crucial if we want to avoid enumerating a large, potentially unbounded, set of witnesses. This can be the case, for example, in line 5 of pushGeneralization. In this particular case, we find a state m at level n (because $m \models F_n$), and a transition t that leads to a problematic clause in F_{n+1}. Therefore we have a sequence σ of size $k - n + 1$ such that $m \overset{\sigma}{\Rightarrow} m'$ and $m' \models \mathbb{F}$. We consider three possible methods for generalizing the trio $(m, \sigma, \mathbb{F})$, that corresponds to property (G1)–(G3) in Lemma 2.

```
Function pushGeneralization($states$: set of (state, level), $k$: level)
while $\top$ do
    $(s, n) \leftarrow$ from $states$ minimizing $n$
    if $n > k$ then
        return
    if $(m \xrightarrow{t} m') \models F_n(\boldsymbol{p}) \wedge T(\boldsymbol{p}, \boldsymbol{p'}) \wedge s(\boldsymbol{p'})$ then
        $p \leftarrow$ generalizeWitness($m$, $t$, $s$)
        $l \leftarrow$ inductivelyGeneralize($p$, $n$ - $2$, $k$)
        $states \leftarrow states \cup \{(p, l+1)\}$
    else
        $l \leftarrow$ inductivelyGeneralize($s$, $n$, $k$)
        $states \leftarrow states \setminus \{(s, n)\} \cup \{(s, l+1)\}$
```

State-based Generalization. A special case of the reachability problem is when the predicate $\mathbb{F}$ is monotonic,, meaning that $m_1 \models \mathbb{F}$ entails $m_1 + m_2 \models \mathbb{F}$ for all markings m_1, m_2. A sufficient (syntactic) condition is for $\mathbb{F}$ to be a positive formula with literals of the form $\sum_{i \in I} p_i \geq a$. This class of predicates coincide with what is called a *coverability property*, for which there exists specialized verification methods (see e.g. [18,19]).

By property (G1), If we have to block a witness m such that $m \overset{\sigma}{\Rightarrow} m'$ and $m' \models \mathbb{F}$, we can as well block all the states greater than m. Hence we can choose the predicate GEQ_m to generalize m. This is a very convenient case for verification and one of the optimizations used in previous works on PDR for Petri nets [1,16,23,24]. First, the generalization is very simple and we can easily compute a MIC when we block predicate GEQ_m in a frame. Also, we can prove the completeness of the procedure when $\mathbb{F}$ is monotonic. An intuition is that it is enough, in this case, to check the property on the minimal coverability set of the net, which is always finite [18]. The procedure is also complete for finite transition systems. These are the only cases where we have been able to prove that our method always terminates.

Transition-based Generalization. We propose a new generalization based on the notion of hurdles. This approach can be used when $\mathbb{F}$ is not monotonic, for example when we want to check an invariant that contains literals of the form $p = k$ (e.g. the reachability of a fixed marking) or $p \geqslant q$.

Assume we need to block a witness of the from $m \overset{\sigma}{\Rightarrow} m' \models s$. Typically, s is a cube in $\mathbb{F}$, or a state resulting from a call to pushGeneralization. By property (G2), we can as well block all the states satisfying $G_\sigma(\boldsymbol{p}) \overset{\text{def}}{=} \text{GEQ}_{H(\sigma)}(\boldsymbol{p}) \wedge s(\boldsymbol{p} + \Delta(\sigma))$. This generalization is interesting when property s does not constraint all the places, or when we have few equality constraints. In this case G_σ may have an infinite number of models. It should be noted that using the duality between "feasible traces" and hurdles is not new. For example, it was used recently [19] to accelerate the computation of coverability trees. Nonetheless, to the best of our knowledge, this is the first time that this generalization method has been used with PDR.

Saturated Transition-based Generalization. We still assume that we start from a witness $m \overset{\sigma}{\Rightarrow} m' \models s$. Our last method relies on property (G3) and allows us to consider several iterations of σ. If we fix the value of k, then a possible generalization is $G^k_\sigma \overset{\text{def}}{=} \big(\bigwedge_{i\in 1..n}(p_i \geqslant a(i) + k \cdot b(i))\big) \wedge s(\boldsymbol{p} + (k+1) \cdot \Delta(\sigma))$, where a, b are the mappings of $\mathbb{N}^P$ defined in Lemma 2. (Notice that $G^1_\sigma = G_\sigma$.) More generally the predicate $G^{\leqslant k}_\sigma = G^1_\sigma \vee \cdots \vee G^k_\sigma$ is a valid generalization for the witness (m, σ, s), in the sense that if $m_1 \models G^{\leqslant k}_\sigma$ then there is a trace $m_1 \rightarrow^\star m_2$ such that $m_2 \models s$. At the cost of using existential quantification (and therefore a "top-level" universal quantification when we negate the predicate to block it in a frame), we can use the more general predicate $G^\star_\sigma \overset{\text{def}}{=} \exists k. G^k_\sigma$, which is still linear and has its support in P.

We know examples of invariants where the PDR method does not terminate except when using saturation. A simple example is the net Parity, used as an example in Sect. 2, with the invariant $\mathbb{P} = (p \geqslant 1)$. In this case, $\mathbb{F} = \neg\mathbb{P} = (p = 0)$. Hence we are looking for witnesses such that $m \rightarrow^\star 0$. The simplest example is $2 \xrightarrow{t_2} 0$, which corresponds to the "blocking clause" $p \neq 2$. In this case, we have $H(t_2) = 2$ and $\Delta(t_2) = -2$. Hence the transition-based generalization is $(p \geq 2) \wedge (p - 2 = 0) \equiv (p = 2)$, which does not block new markings. At this point, we try to block $(p = 0) \vee (p = 2)$. The following minor iteration of our method will consider the witness $4 \xRightarrow{t_2 . t_2} 0$, etc. Hence after k minor iterations, we have $F_k \equiv (p \neq 0) \wedge (p \neq 2) \wedge \cdots \wedge (p \neq 2k)$. If we saturate t_2, we find in one step that we should block $\exists k.(p - 2 \cdot (k + 1) = 0)$. This is enough to prove that $(p \geqslant 1)$ is an invariant as soon as the initial marking is an odd number.

This example proves that PDR is not complete, without saturation, in the general case. We conjecture that it is also the case with saturation. Even though example Parity is extremely simple, it is also enough to demonstrate the limit of our method without saturation. Indeed, when we only allow unquantified linear predicates with variables in P, it is not possible to express all the possible semilinear sets in $\mathbb{N}^P$. (We typically miss some periodic sets.) In practice, it is not always useful to saturate a trace and, in our implementation, we use heuristics to limit the number of quantifications introduced by this operation. Actually, nothing prevents us from mixing our different kinds of generalization together, and there is still much work to be done in order to find good tactics in this case.

4 Experimental Results

We have implemented our complete approach in a tool, called SMPT (for Satisfiability Modulo P/T Nets), and made our code freely available under the GPLv3 license. The software, scripts and data used to perform our analyses are available on Github (htttps://github.com/nicolasAmat/SMPT) and are archived in Zenodo [2]. The tool supports the declaration of reachability constraints expressed

Instance	SMPT	ITS-Tools	LoLA	Tapaal
Murphy	**0.75** *	TLE	TLE	TLE
PGCD	**0.11** *	139.08	TLE	TLE
CryptoMiner	0.19 *	5.92	TLE	**0.18**
Parity	0.40 *	3.36	**0.01**	4.16
Process	83.39	TLE	**0.03**	0.18

Table 1. Computation time on our synthetic examples (time in seconds).

using the same syntax as in the Reachability examinations of the Model Checking Contest (MCC). For instance, we use PNML as the input format for nets. SMPT relies on a SMT solver to answer `sat` and `unsat-core` queries. It interacts with SMT solvers using the SMT-LIBv2 format, which is a well-supported interchange format. We used the z3 solver for all the results presented in this section.

Evaluation on Expressiveness. It is difficult to find benchmarks with unbounded Petri nets. To quote Blondin et al. [7], "due to the lack of tools handling reachability for unbounded state spaces, benchmarks arising in the literature are primarily coverability instances". It is also very difficult to randomly generate a true invariant that does not follow, in an obvious way, from the state equation. For this reason, we decided to propose our own benchmark, made of five synthetic examples of nets, each with a given invariant. This benchmark is freely available and presented as an archive similar to instances of problems used in the MCC.

Our benchmark is made of deceptively simple nets that have been engineered to be difficult or impossible to check with current techniques. Our two first examples are displayed in Fig. 1. We give another example in Fig. 2. Each example is quite small, with less than 10 places or transitions, and is representative of patterns that can make the reachability problem harder: the use of self-loops; dead transitions that cannot be detected with the state equation; weights that are relatively prime; etc.

We compared SMPT against ITS-Tools, LoLA, and Tapaal and give our results in Table 1. All results are computed using 4 cores, a limit of 16 GB of RAM, and a timeout of 1 h. A result of TLE stands for "Time Limit Exceeded". For SMPT, we marked with an asterisk (*) the results computed using our saturation-based generalization. Our results show that SMPT is able to answer on several classes of examples that are out of reach for some, or all the other tools; often by orders of magnitude.

Computing Certificate of Invariance. A distinctive feature of SMPT is the ability to output a linear inductive invariant for reachability problems: when we find that $\mathbb{P}$ is invariant, we are also able to output an inductive formula $\mathbb{C}$, of

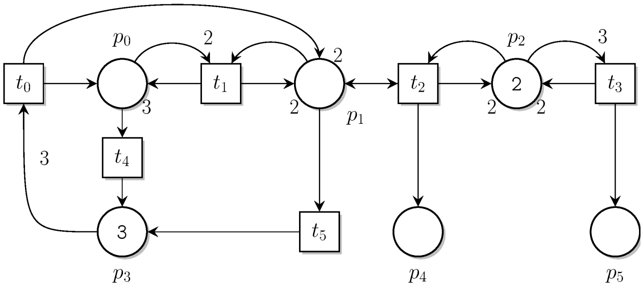


Fig. 2. Example Murphy, with invariant $\mathbb{P} = (p_1 \leqslant 2 \wedge p_4 \geqslant p_5)$.

the form $\mathbb{P} \wedge G$, that can be checked independently with a SMT solver. We can find the same capability in the tool PETRINIZER [16] in the case of coverability properties.

To get a better sense of this feature, we give the actual outputs computed with SMPT on the two nets of Fig. 1. The invariant for the net Parity is $\mathbb{P}_1 = (p_0 \geqslant 1)$, and for PGCD it is $\mathbb{P}_2 = (p_1 \leqslant p_2)$

The certificate for property $\mathbb{P}_1$ on Parity is $\mathbb{C}_1 \equiv (p_0 \geqslant 1) \wedge \forall k.((p_0 < 2\,k + 2) \vee (p_0 \geqslant 2\,k + 3))$, which is equivalent to $(p_0 \geqslant 1) \wedge (\forall k \geqslant 1).(p_0 \neq 2.k)$, meaning the marking of p_0 is odd. This invariant would be different if we changed the initial marking to an even number.

```
[PDR] Certificate of invariance
# (not (p0 < 1))
# (forall (k1) ((p0 < (2 + (k1 * 2))) or (p0 + (-2 * (k1 + 1))) >= 1))
```

The certificate for property $\mathbb{P}_2$ on PGCD is $\mathbb{C}_2 \equiv (p_1 \leqslant p_2) \wedge \forall k.((p_0 < k + 3) \vee (p_2 - p_1 \geqslant k + 1))$ and may seem quite inscrutable. It happens actually that the saturation "learned" the invariant $p_0 + p_1 = p_2 + 2$ and was able to use this information to strengthen property $\mathbb{P}_2$ into an inductive invariant.

```
[PDR] Certificate of invariance
# (not (p1 > p2))
# (forall (k1) ((p0 < (3 + (k1 * 1))) or ((p1 + (1 * (k1 + 1))) <= p2))
```

Evaluation on Performance. Since it is not sufficient to use only a small number of hand-picked examples to check the performance of a tool, we also provide results obtained on a set of 30 problems (a net together with an invariant) that are borrowed from test cases used by the tool SARA [32,33] and a similar software, called REACH, that is part of the TINA toolbox [5]. Most of these problems can be easily answered, but are interesting to test our reliability on a relatively even-handed benchmark.

The experiments were performed with the same conditions as previously. We display our results in the chart of Fig. 3, which gives the number of feasible problems, for each tool, when we change the timeout value. We observe that

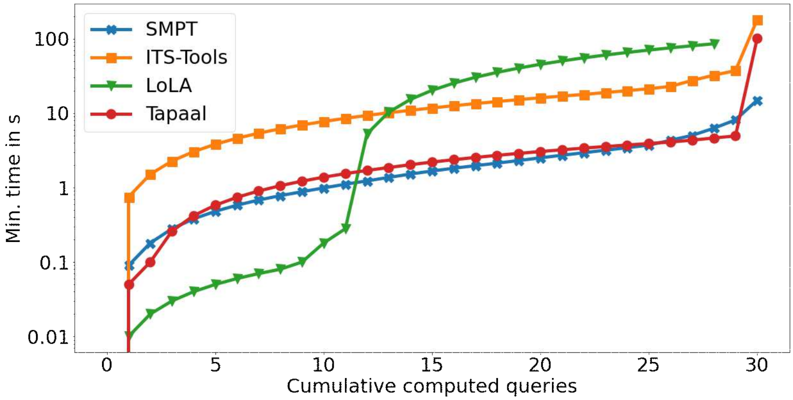


Fig. 3. Minimal timeout to compute a given number of queries.

our performances are on par with TAPAAL, which is the fastest among our three reference tools on this benchmark.

Our tool is actually quite mature. In particular, a preliminary version of SMPT [1] (without many of the improvements described in this work) participated in the 2021 edition of the MCC, where we ranked fourth, out of five competitors, and achieved a reliability in excess of 99.9%. Even if it was with a previous version of our tool, there are still lessons to be learned from these results. In particular, it can inform us on the behavior of SMPT on a very large and diverse benchmark of bounded nets, with a majority of reachability properties that are not invariants.

We can compare our results with those of LOLA, that fared consistently well in the reachability category of the MCC. LOLA is geared towards model checking of finite state spaces, but it also implements semi-decision procedures for the unbounded case. Out of 45 152 reachability queries at the MCC in 2021 (one instance of a net with one formula), LOLA was able to solve 85% of them (38 175 instances) and SMPT only 52% (23 375 instances); it means approximately ×1.6 more instances solved using LOLA than using SMPT. Most of the instances solved with SMPT have also been solved by LOLA; but still 1 631 instances are computed only with our tool, meaning we potentially increase the number of computed queries by 4%. This is quite an honorable result for SMPT, especially when we consider the fact that we use a single technique, with only a limited number of optimizations.

5 Conclusion and Related Works

One of the most important results in concurrency theory is the decidability of reachability for Petri nets or, equivalently, for Vector Addition Systems with

States (VASS) [25]. Even if this result is based on a constructive proof, and its "construction" streamlined over time [26], the classical Kosaraju-Lambert-Mayr-Sacerdote-Tenney approach does not lead to a workable algorithm. It is in fact a feat that this algorithm has been implemented at all, see e.g. the tool KReach [15]. While the (very high) complexity of the problem means that no single algorithm could work efficiently on all inputs, it does not prevent the existence of methods that work well on some classes of problems. For example, several algorithms are tailored for the discovery of counter-examples. We mention the tool FastForward [7] in our experiments, that explicitly targets the case of unbounded nets.

We propose a method that works as well on bounded as on unbounded ones; that behaves well when the invariant is true; and that works with "genuine" reachability properties, and not only with coverability. But there is of course no panacea. Our approach relies on the use of linear predicates, which are incrementally strengthened until we find an invariant based on: the transition relation of the net; the property we want to prove (it is "property-directed"); and constraints on the initial states. This is in line with a property proved by Leroux [26], which states that when a final configuration is not reachable then "*there exist checkable certificates of non-reachability in the Presburger arithmetic.*" Our extension of PDR provides a constructive method for computing such certificates, when it terminates. For our future works, we would like to study more precisely the completeness of our approach and/or its limits.

This is not something new. There are many tools that rely on the use of integer programming techniques to check reachability properties. We can mention the tool Sara [33], that is now integrated inside LoLA and can answer reachability problems on unbounded nets; or libraries like Fast [4], designed for the analysis of systems manipulating unbounded integer variables. An advantage of our method is that we proceed in a lazy way. We never explicitly compute the structural invariants of a net, never switch between a Presburger formula and its representation as a semilinear set (useful when one wants to compute the "Kleene closure" of a linear constraint), ...and instead let a SMT solver work its magic.

We can also mention previous works on adapting PDR/IC3 to Petri nets. A first implementation of SMPT was presented in [1], where we focused on the integration of structural reductions with PDR. This work did not use our abstraction methods based on hurdles and saturation, which are new. We can find other related works, such as [16,23,24]. Nonetheless they all focus on coverability properties. Coverability is not only a subclass of the general reachability problem, it has a far simpler theoretical complexity (EXPSPACE vs NONELEMENTARY). It is also not expressive enough for checking the absence of deadlocks or for complex invariants, for instance involving a comparison between the marking of two places, such as $p < q$. The idea we advocate is that approaches based on the generalization of markings are not enough. This is why we believe that abstractions (G2) and (G3) defined in Lemma 2 are noteworthy.

We can also compare our approach with tools oriented to the verification of bounded Petri nets; since many of them integrate methods and semi-decision procedures that can work in the unbounded case. The best performing tools in this category are based on a portfolio approach and mix different methods. We compared ourselves with three tools: ITS-Tools [30], Tapaal [14] and LoLA [34], that have in common to be the top trio in the Model Checking Contest [3]. (And can therefore accept a common syntax to describe nets and properties.) Our main contribution in this context, and one of our most complex results, is to provide a new benchmark of nets and properties that can be used to evaluate future reachability algorithms "for expressiveness".

The methods closest to ours in these portfolios are Bounded Model Checking and k-induction [28], which are also based on the use of SMT solvers. We can mention the case of ITS-Tools [31], that can build a symbolic over-approximation of the state space, represented as set of constraints. This approximation is enough when it is included in the invariant that we check, but inconclusive otherwise. A subtle and important difference between PDR and these methods is that PDR needs only $2n$ variables (the $\boldsymbol{p}$ and $\boldsymbol{p'}$), whereas we need n fresh variables at each new iteration of k-induction (so kn variables in total). This contributes to the good performances of PDR since the complexity of the SMT problems are in part relative to the number of variables involved. Another example of over-approximation is the use of the so-called "state equation method" [27], that can strengthen the computations of inductive invariants by adding extra constraints, such as place invariants [29], siphons and traps [16,17], causality constraints, etc. We plan to exploit similar constraints in SMPT to better refine our invariants.

To conclude, our experiments confirm what we already knew: we always benefit from using a more diverse set of techniques, and are still in need of new techniques, able to handle new classes of problems. For instance, we can attribute the good results of Tapaal, in our experiments, to their implementation of a Trace Abstraction Refinement (TAR) techniques, guided by counter-examples [10]. The same can be said with LoLA, that also uses a CEGAR-like method [33]. We believe that our approach could be a useful addition to these techniques.

Acknowledgements. We would like to thank Alex Dixon, Philip Offtermatt and Yann Thierry-Mieg for their support when evaluating their respective tools. Their assistance was essential in improving the quality of our experiments.

References

1. Amat, N., Berthomieu, B., Dal Zilio, S.: On the combination of polyhedral abstraction and SMT-based model checking for Petri nets. In: International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets). LNCS, vol. 12734. Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_9
2. Amat, N., Dal Zilio, S., Hujsa, T.: SMPT (2022). https://doi.org/10.5281/zenodo.5863379

3. Amparore, E., Berthomieu, B., Ciardo, G., Dal Zilio, S., Gallà, F., Hillah, L.M., Hulin-Hubard, F., Jensen, P.G., Jezequel, L., Kordon, F., Le Botlan, D., Liebke, T., Meijer, J., Miner, A., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y., van Dijk, T., Wolf, K.: Presentation of the 9th edition of the model checking contest. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer (2019). https://doi.org/10.1007/978-3-662-58381-4_9
4. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: acceleration from theory to practice. International Journal on Software Tools for Technology Transfer **10**(5) (2008). https://doi.org/10.1007/s10009-008-0064-3
5. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets. International journal of production research **42**(14) (2004)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
7. Blondin, M., Haase, C., Offtermatt, P.: Directed reachability for infinite-state systems. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_1
8. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 6538. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
9. Bradley, A.R.: Understanding IC3. In: Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 7317. Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_1
10. Cassez, F., Jensen, P.G., Larsen, K.G.: Refinement of trace abstraction for real-time programs. In: International Workshop on Reachability Problems. Springer (2017). https://doi.org/10.1007/978-3-319-67089-8_4
11. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_4
12. Czerwiński, W., Lasota, S., Lazić, R., Leroux, J., Mazowiecki, F.: The reachability problem for Petri nets is not elementary. Journal of the ACM (JACM) **68**(1) (2020). https://doi.org/10.1016/0304-3975(79)90041-0
13. Czerwinski, W., Orlikowski, L.: Reachability in vector addition systems is Ackermann-complete. CoRR **abs/2104.13866** (2021), https://arxiv.org/abs/2104.13866
14. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_36
15. Dixon, A., Lazić, R.: Kreach: A tool for reachability in Petri nets. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 12078. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_22
16. Esparza, J., Ledesma-Garza, R., Majumdar, R., Meyer, P., Niksic, F.: An SMT-Based Approach to Coverability Analysis. In: Computer Aided Verification (CAV). LNCS (2014). https://doi.org/10.1007/978-3-319-08867-9_40
17. Esparza, J., Melzer, S.: Verification of safety properties using integer programming: Beyond the state equation (2000). https://doi.org/10.1023/A:1008743212620

18. Finkel, A.: The minimal coverability graph for Petri nets. In: International Conference on Application and Theory of Petri Nets. Springer (1991). https://doi.org/10.1007/3-540-56689-9_45
19. Finkel, A., Haddad, S., Khmelnitsky, I.: Commodification of accelerations for the Karp and Miller construction. Discret. Event Dyn. Syst. **31**(2) (2021). https://doi.org/10.1007/s10626-020-00331-z
20. Ginsburg, S., Spanier, E.: Semigroups, Presburger formulas, and languages. Pacific journal of Mathematics **16**(2) (1966). https://doi.org/10.2140/pjm.1966.16.285
21. Hack, M.H.T.: Decidability questions for Petri Nets. Ph.D. thesis, Massachusetts Institute of Technology (1976)
22. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: International Conference on Theory and Applications of Satisfiability Testing (SAT). Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_13
23. Kang, J., Bai, Y., Jiao, L.: Abstraction-based incremental inductive coverability for Petri nets. In: International Conference on Applications and Theory of Petri Nets and Concurrency. LNCS, vol. 12734. Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_19
24. Kloos, J., Majumdar, R., Niksic, F., Piskac, R.: Incremental, inductive coverability. In: Computer Aided Verification (CAV). Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_10
25. Kosaraju, S.R.: Decidability of reachability in vector addition systems. In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing. ACM (1982). https://doi.org/10.1145/800070.802201
26. Leroux, J.: The general vector addition system reachability problem by Presburger inductive invariants. In: 2009 24th Annual IEEE Symposium on Logic In Computer Science. IEEE (2009). https://doi.org/10.1109/LICS.2009.10
27. Murata, T.: State equation, controllability, and maximal matchings of petri nets. IEEE Transactions on Automatic Control **22**(3) (1977). https://doi.org/10.1109/TAC.1977.1101509
28. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Formal Methods in Computer-Aided Design. LNCS, Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
29. Silva, M., Terue, E., Colom, J.M.: Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In: Advanced Course on Petri Nets. Springer (1998). https://doi.org/10.1007/3-540-65306-6_19
30. Thierry-Mieg, Y.: Symbolic Model-Checking Using ITS-Tools. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_20
31. Thierry-Mieg, Y.: Structural reductions revisited. In: Application and Theory of Petri Nets and Concurrency. LNCS, vol. 12152. Springer (2020). https://doi.org/10.1007/978-3-030-51831-8_15
32. Wimmel, H.: Sara: Structures for automated reachability analysis (2013), https://github.com/nlohmann/service-technology.org/tree/master/sara
33. Wimmel, H., Wolf, K.: Applying CEGAR to the Petri net state equation. Logical Methods in Computer Science **8** (2012). https://doi.org/10.2168/LMCS-8(3:27)2012
34. Wolf, K.: Petri net model checking with LoLA 2. In: Application and Theory of Petri Nets and Concurrency. Springer (2018). https://doi.org/10.1007/978-3-319-91268-4_18

Transition Power Abstractions for Deep Counterexample Detection*

Martin Blicha[1,3](✉), Grigory Fedyukovich[2], Antti E.J. Hyvärinen[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Lugano, Switzerland, first.last@usi.ch
[2] Florida State University, Tallahassee, FL, USA, grigory@cs.fsu.edu
[3] Charles University, Prague, Czech Republic

Abstract. While model checking safety of infinite-state systems by inferring state invariants has steadily improved recently, most verification tools still rely on a technique based on bounded model checking to detect safety violations. In particular, the current techniques typically analyze executions by unfolding transitions one step at a time, and the slow growth of execution length prevents detection of deep counterexamples before the tool reaches its limits on computations. We propose a novel model-checking algorithm that is capable of both proving unbounded safety and finding long counterexamples. The idea is to use Craig interpolation to guide the creation of symbolic abstractions of *exponentially longer sequences of transitions*. Our experimental analysis shows that on unsafe benchmarks with deep counterexamples our implementation can detect faulty executions that are at least an order of magnitude longer than those detectable by the state-of-the-art tools.

Keywords: Model checking · Transition systems · Craig interpolation · Model-based projection.

1 Introduction

Model checking [17] is a very successful technique widely used for formal verification of hardware and software. While its ultimate goal is to *prove* safety, the ability to discover and report counterexamples primarily contributes to its industrial success. The algorithm that paved the way for the adaptation in the industry, *bounded model checking* (BMC) [9], still remains one of the most successful techniques today for detecting counterexamples. A typical BMC algorithm searches for counterexamples reachable in a finite number of steps, and if nothing is found, it increases the search limits and restarts. This philosophy has been largely adopted by most modern model-checking algorithms based on reachability

* The first author is partially funded by the project 20-07487S of the Czech Science Foundation. The first, third, and forth authors are partially funded by the Swiss National Science Foundation project 200021_185031. The second author is partially funded by the gift from Amazon Web Services.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 524–542, 2022.
https://doi.org/10.1007/978-3-030-99524-9_29

analysis as one of the advantages of this approach is that it finds the shortest counterexample (if one exists). However, it also results in scalability issues. Specifically, in modern software systems, it is not uncommon that a program must iterate through a certain loop thousands of times (or more) before it reaches some error state. These *deep* counterexamples pose problems for reachability-based algorithms that rely on unrolling the bounds of the system's transition relation one transition at a time.

An important class of loops present in software systems are *multi-phase* loops [44]. A multi-phase loop, in short, is a loop with a conditional (branch) in its body such that the conditional exhibits a fixed number of phase transitions during the execution of the loop. A phase is a sequence of iterations during which the conditional has the same value. Multi-phase loops are notoriously challenging to analyze. When they are safe, they typically require disjunctive invariants. On the other hand, an unsafe multi-phase loop may admit only deep counterexamples if only later phases reveal the unsafe behavior.

In this paper we present a novel model-checking algorithm that is able to find counterexamples of much greater depth than state-of-the-art algorithms. At the same time, it is able to prove system safe under certain conditions and is competitive also on a general set of benchmarks. We build upon the large body of work on SMT-based model checking [1,3,4,8,14,15,25,28,30,37,38] and use Craig interpolation [18,35] for computing abstractions. However, we shift the focus from *state* abstractions—which is the widespread approach—to *transition* abstractions [40].

Our algorithm works on transition systems and it builds a sequence of abstract relations that gradually summarize (in an over-approximating way) an increasing number of steps of the transition relation. One important feature is that the summarized number of steps increases exponentially, not linearly. Another important feature is that all the abstract relations are expressed only over state and next-state variables, i.e., they do not require multiple copies of state variables to capture multiple steps of the transition relation. This sequence of abstract relations is used to refute the existence of bounded reachability paths in the system. If existence of a path cannot be refuted in the current abstraction, either the abstraction is strengthened to refute such path, or the path is shown to be real. The precise mechanics of building and refining the sequence of abstract relations are explained in Section 4. Our experiments demonstrate that our algorithm improves the ability to detect deep counterexamples in the multi-phase loop programs up to two orders of magnitude compared to the state-of-the-art. Furthermore, it enables the detection of bugs left undiscovered by the other tools.

The main contributions of the paper are the following:

- A novel model-checking algorithm for safety properties of transition system based on a sequence of relations over-approximating exponentially increasing number of steps of transition relation.
- Proof of correctness of the algorithm and its termination for unsafe systems.

- Implementation and experimental evaluation of the proposed algorithm demonstrating its capabilities of finding deep counterexamples in challenging benchmarks containing multi-phase loops.

The rest of the paper is organized as follows. The necessary background is given in Section 2, and a motivating example is given in Section 3. Section 4 describes our novel algorithm, and Section 5 presents the experimental results. We discuss the related work in Section 6 and conclude in Section 7.

2 Background

Safety problem We work with a standard symbolic representation of transition systems using the language of first-order logic. Given a set of variables X, we denote as X' the primed copy of X, i.e., $X' = \{x' \mid x \in X\}$. X is a set of *state* variables and X' is a set of *next-state* variables. The formulas are interpreted with respect to some background theory $\mathcal{T}$; in our examples and benchmarks we work with the theory of linear real or integer arithmetic (LRA and LIA in the terminology of satisfiability modulo theories (SMT) [6,7]). We say that a formula in the language of $\mathcal{T}$ over X is a *state* formula and a formula over $X \cup X'$ is a *transition* formula. We identify state formulas with a set of states where they hold and we freely move between these two representations. Similarly, we identify transition formulas with binary relations over the set of states. The identity relation $Id(x, x')$ corresponds to the transition formula $x = x'$.

Transition system is a pair $\langle Init, Tr \rangle$ where $Init$ is a state formula representing the initial states of the system and Tr is a transition formula representing the transition relation of the system. A *safety problem* is a triple $\langle Init, Tr, Bad \rangle$ where $\langle Init, Tr \rangle$ is a transition system and Bad is a state formula representing bad states.

When we only need to distinguish state and next-state variables, but not the individual state variables, for simplicity we only use the lower-case x, x' and not X, X'. These can be viewed as variables representing tuples. We also often need to refer to next-next-state variables, which we denote as x''.

We use $\circ$ to represent *concatenation* of relations. For example, given two relations $R_1(x, y)$ and $R_2(y, z)$ then $R = R_1 \circ R_2$ is a relation over x, z such that $R(x, z) \iff \exists y : R_1(x, y)$ and $R_2(y, z)$. In transition systems we can define relations that represent multiple steps of a transition relation. For example $Tr^2(x, x'') \equiv Tr(x, x') \circ Tr(x', x'')$ relates pair of states (s, t) such that t is reachable from s in *exactly* two steps of the transition relation Tr. We also write that $(s, t) \in Tr^2$. Existence of a counterexample (a path from some initial to some bad state) of a fixed length l can be encoded as a satisfiability check of formula

$$Init(x^{(0)}) \wedge Tr(x^{(0)}, x^{(1)}) \wedge Tr(x^{(1)}, x^{(2)}) \wedge \ldots \wedge Tr(x^{(l-1)}, x^{(l)}) \wedge Bad(x^{(l)}),$$

where $x^{(i)}$ is a state variable shifted i steps, "with i primes". A satisfying assignment determines $l + 1$ states such that the first one is an initial state, the last one is a bad state, and each successor can be reached from its predecessor by one step of the transition relation *Tr*. If there is no satisfying assignment then no path of l steps from *Init* to *Bad* exists.

Craig interpolation [18] Given an unsatisfiable formula $A \wedge B$, an interpolant I is a formula over the shared symbols of A and B such that $A \implies I$ and $I \wedge B$ is unsatisfiable. We denote as $\mathit{Itp}(A, B)$ an interpolation procedure that computes an interpolant for unsatisfiable $A \wedge B$. Various interpolation procedures exist, for propositional logic [31,42,34,19] as well as for different first-order theories [36,16,2,11].

3 Motivating example

Throughout the paper we demonstrate our approach on a family of C-like programs with a multi-phase loop (generalized from [44] where `N=50`) and an unsafe assertion. The use of parameter `N` (should not be confused with a nondeterministic variable) demonstrates the scale of search of counterexamples of different lengths. We have experimentally evaluated how various tools perform on this example in Section 5. The program source code and the corresponding transition system are given in Figure 1.

```
x=0; y=N;
while(x < 2N){
  x = x + 1;
  if(x > N)
    y = y + 1;
}
assert(y != 2N);
```

$$\begin{aligned}
\mathit{Init}(x, y) &\equiv x = 0 \wedge y = N \\
\mathit{Tr}(x, y, x', y') &\equiv x < 2N \wedge x' = x + 1 \\
&\quad \wedge y' = \mathit{ite}(x' > N, y + 1, y) \\
\mathit{Bad}(x, y) &\equiv x \geq 2N \wedge y = 2N
\end{aligned}$$

Fig. 1: An example of unsafe multi-phase loop

Since the assertion is placed after the loop, any counterexample requires finding a complete unrolling of the loop, i.e., all `2N` iterations (or `2N` steps in the corresponding transition system). Interestingly, even a linear growth of `N` results in the exponential growth of complexity of search of counterexamples. Because of the control-flow divergence in each iteration of the loop, the number of possible program paths (that a verifier explores) doubles with each increment of counter `x`. Our technique allows finding the counterexamples for any `N` drastically more efficiently.

```
input  : transition system $\mathcal{S} = \langle \mathit{Init}, \mathit{Tr}, \mathit{Bad} \rangle$
global : TPA sequence $S$ (lazily initialized to true)
Function CheckSafetyTPA($\langle \mathit{Init}, \mathit{Tr}, \mathit{Bad} \rangle$):
    $S[0] \leftarrow \mathit{Id} \vee \mathit{Tr}$
    if $\mathit{Sat}?[\mathit{Init}(x) \wedge S[0](x, x') \wedge \mathit{Bad}(x')]$ then return UNSAFE
    $n \leftarrow 0$
    while $\mathit{TRUE}$ do
        $\mathit{res} \leftarrow$ IsReachable($n, \mathit{Init}, \mathit{Bad}$)
        if $\mathit{res} \neq \emptyset$ then return UNSAFE
        $n \leftarrow n + 1$
    end
```

Algorithm 1: Main procedure for checking safety

4 Finding deep counterexamples with transition power abstractions

Our main procedure for detecting safety violation—given in Algorithm 1—follows the typical scheme of bounded model checking where in each iteration the reachability of *Bad* is checked within certain *bounded* number of steps and the bound gradually increases. This scheme has also been adopted by other model checking algorithms, such as Spacer [30] and interpolation-based model checking [20,34,45], which further support a generalization/adaptation of the proof of bounded safety to a proof of unbounded safety.

The distinguishing feature of our approach is that it increases the bound for the safety check *exponentially* in the number of iterations, while other approaches do this linearly. That is, in the n^{th} iteration, traditional algorithms check bounded safety up to n steps; but our approach does up to 2^{n+1} steps. However, we do *not* unroll the transition relation an exponential number of times. Instead, we maintain a sequence of transition formulas (i.e., each formula contains only *two* copies of the state variables) where each element over-approximates twice as many steps of transition relation Tr as its predecessor. We call this sequence a *Transition Power Abstraction* (TPA) sequence.

4.1 TPA sequence for bounded reachability queries

The core of our approach lies in *creating and refining* a sequence of relations $\mathit{ATr}^{\leq 0}, \mathit{ATr}^{\leq 1}, \ldots, \mathit{ATr}^{\leq n}, \ldots$ where each relation over-approximates *twice* as many transition steps of a transition relation Tr as its predecessor. Formally, we require that n^{th} relation $\mathit{ATr}^{\leq n}$ satisfies:

$$\mathit{Id}(x, x') \vee \mathit{Tr}(x, x') \vee \mathit{Tr}^2(x, x') \vee \ldots \vee \mathit{Tr}^{2^n}(x, x') \implies \mathit{ATr}^{\leq n}(x, x') \qquad (1)$$

The base for constructing a TPA sequence is $\mathit{ATr}^{\leq 0} \equiv \mathit{Id} \vee \mathit{Tr}$. Thus, $\mathit{ATr}^{\leq 0}$ is *not* an over-approximation, but a precise relation capturing true reachability in either 0 or 1 steps.

Our check for bounded safety is based on a procedure that answers *bounded reachability queries*: Given a set of *source* and *target* states, is any target state reachable from some source state in up to 2^{n+1} steps (for $n \geq 0$)? The procedure *uses* the TPA sequence to answer such queries and, at the same time, it *extends* the sequence and *refines* its existing elements.

Given two sets of states, *Source* and *Target*, and n^{th} element of the current TPA sequence $ATr^{\leq n}$, the following SMT query is issued:

$$Sat?[Source(x) \wedge ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'') \wedge Target(x'')]. \tag{2}$$

If query (2) is unsatisfiable, it means that there is no *intermediate* state that would be reachable from *Source* using one step of $ATr^{\leq n}$ and, at the same time, can reach *Target* in yet another step of $ATr^{\leq n}$. Since one step of $ATr^{\leq n}$ over-approximates reachability (using Tr) in 0 to 2^n steps, this means that no path of length $\leq 2^{n+1}$ exists from *Source* to *Target*. Thus, the procedure can immediately conclude that no state from *Target* is reachable from any state in *Source* in $\leq 2^{n+1}$ steps.

Additionally, it is also possible to learn new information about the reachability in $\leq 2^{n+1}$ steps in the form of an interpolant between $ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'')$ and $Source(x) \wedge Target(x'')$. The properties of interpolation guarantee that the interpolant contains only variables x, x'' (i.e., it does not contain x'), it over-approximates $ATr^{\leq n} \circ ATr^{\leq n}$, and it does not relate any source state with a target state. The relation defined by such an interpolant satisfies condition (1) for the $n{+}1^{\text{st}}$ element of TPA sequence and the current TPA sequence can be refined by conjoining the interpolant (after renaming of variables) to its $n{+}1^{\text{st}}$ element.

If query (2) is satisfiable, there exists some *intermediate* state m that can be reached from *Source* by one step of $ATr^{\leq n}$ and that can reach *Target* by yet another step of $ATr^{\leq n}$. If $n = 0$, the procedure returns and reports the answer "reachable" as $ATr^{\leq 0}$ is *precise*, not over-approximating. Otherwise, such an intermediate state m can be seen as a potential point on the path from *Source* to *Target*, and this path can be shown to be real if there exist two real paths: from *Source* to m and from m to *Target*. The existence of these two real paths can be checked in a recursive manner.

4.2 Algorithm for bounded reachability checks

The pseudocode for the procedure is given in Algorithm 2. We first explain the steps in more detail and demonstrate a run of the algorithm on our example from Section 3. We then prove the correctness and termination of Algorithm 2 from which follow the correctness of Algorithm 1 and its termination for unsafe systems.

Function `IsReachable` takes as input an integer $n \geq 0$, a set of source states, and a set of target states. The output is a subset of target states that are reachable in $\leq 2^{n+1}$ steps of transition relation Tr. The output set is empty if and only if no target state is reachable from any source state within the given bound.

```
input  : level $n$, source states $Source$, target states $Target$
output : subset of $Target$ reachable from $Source$ within $2^{n+1}$ steps
global : TPA sequence $S$
Function IsReachable($n$,$Source$,$Target$):
1   while $true$ do
2       $ATr^{\leq n} \leftarrow S[n]$
3       $query \leftarrow Source(x) \wedge ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'') \wedge Target(x'')$
4       $sat\_res \leftarrow Sat?[query]$
5       if $sat\_res = UNSAT$ then
6           $I \leftarrow Itp(ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x''), Source(x) \wedge Target(x''))$
7           $S[n + 1] \leftarrow S[n + 1] \wedge I[x'' \mapsto x']$
8           return $\emptyset$
9       else
10          if $n = 0$ then return $QE(\exists x, x' \; query)[x'' \mapsto x]$
11          $Intermediate \leftarrow QE(\exists x, x'' \; query)[x' \mapsto x]$
12          $IntermediateReached \leftarrow$ IsReachable($n - 1, Source, Intermediate$)
13          if $IntermediateReached = \emptyset$ then continue
14          $TargetReached \leftarrow$ IsReachable($n - 1, IntermediateReached, Target$)
15          if $TargetReached = \emptyset$ then continue
16          return $TargetReached$
17      end
18  end
```

Algorithm 2: Reachability query using TPA

The procedure loops until it computes a truly reachable subset of target states or proves all target states unreachable. In each iteration the procedure reads the current n^{th} element of the TPA sequence (line 2). Note that this will be different in each iteration as the TPA sequence will be updated in the recursive calls on lines 13 and 15. After that, a satisfiability query is constructed and passed to a decision procedure for the background theory $\mathcal{T}$ (lines 3 and 4). The satisfiability query represents a question whether or not there exists an intermediate state that would be reachable from $Source$ using one step of $ATr^{\leq n}$ and, at the same time, can reach $Target$ in yet another step of $ATr^{\leq n}$.

Query on line 4 is unsatisfiable. If the query is unsatisfiable then no target state can be reached from any source state in two steps of $ATr^{\leq n}$. It follows from Eq. (1) that no target state can be reached from any source state in $\leq 2^{n+1}$ steps. Before indicating the unreachability by returning $\emptyset$ (line 8), the function updates the TPA sequence to ensure termination (discussed later): The function computes an interpolant between $ATr^{\leq n}(x, x') \wedge ATr^{\leq n}(x', x'')$ and $Source(x) \wedge Target(x'')$ (line 6). After renaming variables, the interpolant is conjoined to the $n+1^{\text{st}}$ element of the TPA sequence. The following example demonstrates this part of the procedure on our motivating example.

Example 1. Consider the system from Figure 1 for $N = 3$. This system is not safe and the counterexample requires six steps of transition relation Tr.

After Algorithm 1 initializes the base element of TPA sequence to $(x' = x \wedge y' = y) \vee (x < 6 \wedge x' = x + 1 \wedge y' = ite(x' > 3, y + 1, y))$ it issues a reachability query $\texttt{IsReachable}(0, x = 0 \wedge y = 3, x \geq 6 \wedge y = 6)$ in the first iteration of its loop. This translates to a satisfiability check of the formula

$$\begin{aligned}
&x = 0 \wedge y = 3\\
&\wedge ((x' = x \wedge y' = y) \vee (x < 6 \wedge x' = x + 1 \wedge y' = ite(x' > 3, y + 1, y)))\\
&\wedge ((x'' = x' \wedge y'' = y') \vee (x' < 6 \wedge x'' = x' + 1 \wedge y'' = ite(x'' > 3, y' + 1, y')))\\
&\wedge x'' \geq 6 \wedge y'' = 6
\end{aligned}$$

on line 4 of Algorithm 2. This query is unsatisfiable, and $x'' \leq x + 2$ is a possible interpolant computed on line 6. After variable renaming, this interpolant refines $S[1]$, which becomes $x' \leq x + 2$. Then this call to IsReachable terminates and the main loop issues a new reachability query for $n = 1$. This yields a satisfiability query $x = 0 \wedge y = 3 \wedge x' \leq x + 2 \wedge x'' \leq x' + 2 \wedge x'' \geq 6 \wedge y'' = 6$. Again, this formula is unsatisfiable and a possible interpolant is $x'' \leq x + 4$. The next element of the TPA sequence, $S[2]$ is refined to $x' \leq x + 4$.

For $n = 2$ (reachability within eight steps), the query on line 4 is satisfiable, and the procedure switches to checking if the counterexample from abstract transition is real or exists only due to a coarse abstraction.

Query on line 4 is satisfiable. If the query on line 9 is satisfiable, a concrete path of length $\leq 2^{n+1}$ cannot be ruled out at this point and the algorithm proceeds to recursively check the existence of one. In the base case $n = 0$ of the recursion, $ATr^{\leq 0}$ is not an over-approximation but a precise relation representing 0 or 1 steps of Tr and there exists a real path from *Source* to *Target*. The algorithm computes a state formula representing a truly reachable subset of *Target*. This is done by first using *quantifier elimination* (QE) to eliminate all except next-next state variables from the query (line 10) and then renaming the variables to state variables.[4]

If the base case has not been reached yet ($n > 0$), the procedure first computes a set of candidate *intermediate* states by eliminating all except next-state variables from the query (line 11). Then, the procedure recursively calls itself to determine the existence of a path from *Source* to the newly computed intermediate set with the bound on length halved (line 12). This check has two possible outcomes. In case the recursive call returns $\emptyset$, none of the intermediate candidates is reachable (within 2^n steps). Moreover, $S[n]$ must have been strengthened (line 7) before the recursive call returned as to *not* relate any of the source states and intermediate candidates. The procedure then continues to the next iteration (line 13) where it tries to find new intermediate candidates or prove there are none anymore. In case the set returned on line 12 is non-empty, it represents a set of states reachable from *Source* within 2^n steps of Tr. The procedure proceeds to check the existence

[4] QE computes maximal reachable subsets. While this is convenient for proving termination of Algorithm 2, in practice quantifier elimination is a very expensive operation. Our implementation therefore supports also the use of *model-based projection* to efficiently under-approximate quantifier elimination (see Section 4.4).

of a path from these states to the target states (line 14). The reasoning here is the same as for the first recursive call: If *Target* is not reachable, the procedure attempts to find new intermediate candidates in a new iteration. Otherwise, real path from *Source* to *Target* exists and the computed truly reachable states are returned. The returned states are reachable with 2^{n+1} steps as both recursive calls check reachability within 2^n steps.

We continue Example 1 to illustrate this phase of Algorithm 2.

Example 2. Following Example 1, the algorithm is checking bounded reachability between *Init* and *Bad* for $n = 2$, i.e., within 8 steps. The issued satisfiability query is $x = 0 \wedge y = 3 \wedge x' \leq x + 4 \wedge x'' \leq x' + 4 \wedge x'' \geq 6 \wedge y'' = 6$. Eliminating all except next-state variables yields $x' \leq 4 \wedge x' \geq 2$. This results in the call `IsReachable`$(1, x = 0 \wedge y = 3, x \leq 4 \wedge x \geq 2)$. The satisfiability query issued next is $x = 0 \wedge y = 3 \wedge x' \leq x + 2 \wedge x'' \leq x' + 2 \wedge x'' \leq 4 \wedge x'' \geq 2$. This is again satisfiable and yields $x' \leq 2 \wedge x' \geq 0$ after quantifier elimination. Now we reach level 0 with a call `IsReachable`$(0, x = 0 \wedge y = 3, x \leq 2 \wedge x \geq 0)$. The constructed satisfiability query is again satisfiable and since we are at level 0, the procedure returns a set of states truly reachable from $x = 0 \wedge y = 3$ within 2 steps. These can be characterized as $(x = 0 \vee x = 1 \vee x = 2) \wedge y = 3$. The reachable states are reported to level 1 which issues reachability query for the second part: `IsReachable`$(0, (\mathrm{x} = 0 \vee \mathrm{x} = 1 \vee \mathrm{x} = 2) \wedge y = 3, x \leq 4 \wedge x \geq 0)$. This is also successful and returns reachable states $(x = 0 \vee x = 1 \vee x = 2 \vee x = 3 \vee x = 4) \wedge$ $\mathrm{y} = 3$. These are states reachable from *Init* within 4 steps and they are reported to level 2. There, the second part of the counterexample is found in a similar way and the procedure concludes that *Bad* is truly reachable from *Init* within 8 steps.

The behaviour of the algorithm on these examples can be generalized for the system of Figure 1 for larger values of N. The length of the counterexample is $2N$ and let l denote $\lfloor log_2(2N) \rfloor$. The bounded safety will be quickly determined up to 2^l steps with l calls to `IsReachable` which all return $\emptyset$ in their first iteration. On the next iteration, for $n = l$, `IsReachable` will find the real counterexample, but it requires $O(2^l)$ recursive calls to find the counterexample of length in the interval $(2^l, 2^{l+1}]$.

4.3 Correctness and termination

We first prove correctness and termination of Algorithm 2 which then entails correctness of Algorithm 1 and its termination for unsafe systems. We prove the correctness of procedure `IsReachable` separately for the unreachable and the reachable case.

Lemma 1. *If* `IsReachable`*(n, Source, Target) returns $\emptyset$, then no state from Target can be reached from Source within 2^{n+1} steps.*

Proof. The proof relies on the invariant that S is always a TPA sequence, i.e., its elements satisfy the property of Eq. (1). This is obviously true when S is initialized in Algorithm 1. The only update of S happens in Algorithm 2 on line 7.

Consider an update on any level $k \leq n$. From the properties of interpolation, we know that $I(x, x'')$ (on line 6) over-approximates $ATr^{\leq k}(x, x') \wedge ATr^{\leq k}(x', x'')$, which represents two steps of the relation $ATr^{\leq k}$. Since $ATr^{\leq k}$ over-approximates $\leq 2^k$ steps of Tr, it follows that $I(x, x'')$ over-approximates $\leq 2^{k+1}$ steps of Tr. Thus, conjoining it to $ATr^{\leq k+1}$ preserves the condition of Eq. (1).

It follows from Eq. (1) that when the query on line 4 is unsatisfiable, there exists no path of length $\leq 2 \times 2^n = 2^{n+1}$ from any source state to any target state. □

Lemma 2. *If* `IsReachable`*(n, Source, Target) returns a non-empty set Res, then Res $\subseteq$ Target and every state in Res can be reached from some state in Source in $\leq 2^{n+1}$ steps.*

Proof. The proof is by induction on n.

Base case: For $n = 0$ $ATr^{\leq 0}$ represents precise reachability in 0 or 1 step. It follows that if the query on line 4 is satisfiable, some target states are truly reachable from the set of source states in ≤ 2 steps. Moreover, the properties of QE guarantee that $Res = QE(\exists x, x' \; query)[x'' \mapsto x]$ is a subset of $Target(x)$ that are reachable from $Source$ using $ATr^{\leq 0} \circ ATr^{\leq 0}$.

Inductive case: Suppose the claim holds for $n - 1$. If at level n the procedure returned a non-empty set, it must have been the case that the first recursive call (line 12) returned a non-empty set *IntermediateReached* of states truly reachable from *Source* in $\leq 2^n$ steps, by our induction hypothesis. Additionally, the second recursive call (line 14) also returned a non-empty set *TargetReached* that, according to our induction hypothesis, is a subset of *Target* truly reachable from *IntermediateReached* in $\leq 2^n$ steps. It follows that *TargetReached* is a subset of *Target* truly reachable from *Source* in $\leq 2^{n+1}$ steps. □

The correctness of procedure `IsReachable` extends naturally to the correctness of our main procedure.

Theorem 1 (Correctness). *If Algorithm 1 returns UNSAFE, then the system $\mathcal{S}$ is unsafe, i.e., some bad state is reachable from some initial state.*

Proof. The satisfiablity query on line 2 of Algorithm 1 checks reachability in 0 and 1 step. If this query is satisfiable, there exists a counterexample path of length 0 or 1 from some initial state to a bad state.

Otherwise, it enters the loop where UNSAFE is returned only if `IsReachable` returns non-empty set of states for some n. From the correctness of `IsReachable` it follows that the returned set is a subset of *Bad* that is reachable from *Init* in $\leq 2^{n+1}$ steps. Thus there exists a counterexample path in the system. □

Next, we want to show that if there exists a counterexample path in the system, our procedure will eventually report it. This boils down to the question of termination of a single call to `IsReachable`.

Lemma 3. *Assume that the satisfiability check (line 4) terminates, i.e., that the background theory $\mathcal{T}$ is decidable, and that $\mathcal{T}$ has procedures for interpolation and quantifier elimination.*[5] *Then a single call to* `IsReachable` *always terminates.*

[5] The linear arithmetic theories of our experiments satisfy these assumptions.

Proof. The proof proceeds by induction on level n. The base case ($n = 0$) trivially terminates after a single satisfiability query on line 4.

For the inductive case, consider the first iteration of the loop. If the query is unsatisfiable, the procedure terminates. If it is satisfiable, quantifier elimination yields a set of states $\mathit{Intermediate} = \{m \mid \exists s \in \mathit{Source}, \exists t \in \mathit{Target} : (s, m) \in \mathit{ATr}^{\leq n} \wedge (m, t) \in \mathit{ATr}^{\leq n}\}$. Now consider the first recursive call (line 12). By induction, it terminates. If it returns $\emptyset$, then, by properties of the interpolation, $\mathit{ATr}^{\leq n}$ has been strengthened such that $\forall s \in \mathit{Source}, \forall m \in \mathit{Intermediate} : (s, m) \notin \mathit{ATr}^{\leq n}$ now holds. Consequently, in the second iteration the query on line 4 must be unsatisfiable and the procedure terminates.

Now consider the situation where the recursive call on line 12 returned a non-empty set *IntermediateReached*. The procedure continues to the second recursive call (line 14), which also terminates, by induction. If the returned set *TargetReached* is non-empty, the procedure terminates (line 16). If it is empty, then no state reachable from *Source* in $\leq 2^n$ steps of *Tr* can reach any state in *Target* in another $\leq 2^n$ steps. Moreover, $\mathit{ATr}^{\leq n}$ has been strengthened so that now it does not relate any state from *IntermediateReached* with a state in *Target*. In the second iteration, the query on line 4 could still be satisfiable. However, the extracted *Intermediate* (of the second iteration) cannot contain states that are reachable from *Source* in $\leq 2^n$ steps. Thus first recursive call (line 12) in the second iteration must return $\emptyset$ and this is followed by an unsatisfiable query (line 4) in the third iteration and termination. □

The immediate consequence of Lemma 3 is that our main procedure will find a counterexample if one exists.

Theorem 2. *If there exists a counterexample in the system, Algorithm 1 terminates with UNSAFE result.*

4.4 Under-approximating QE with model-based projection

Model-based projection (MBP) [30] is a recent technique for under-approximating quantifier elimination for existentially quantified formulas. In short, given an existentially quantified formula $\exists x \phi(x, y)$, MBP is a function that maps each model of ϕ to a quantifier-free formula that implies $\exists x \phi(x, y)$ and is true in the model. Moreover, it is required that the function has a finite image (it produces only finitely many quantifier-free under-approximations) and the disjunction of the image is equal to the quantified formula. Efficient MBP for linear real and integer arithmetic was given in [30,10]. MBP has also been designed for algebraic datatypes [10], arithmetic signature of bit-vectors [23] and arrays[6] [29].

Quantifier elimination in Algorithm 2 can be replaced by MBP in a straightforward way. On line 4, if the query is satisfiable, we obtain from the SMT solver a model witnessing the satisfiability. Then, on lines 10 and 11 we replace QE with MBP using the obtained model. It is easy to check that the proof of Lemma 2 remains valid with this change, and thus also the result of Theorem 1. In Section 5 we experimentally demonstrate the practical advantage of MBP over QE.

[6] MBP for arrays does not satisfy the finite image condition

4.5 Proving safety

Even though the main purpose of the TPA sequence is to help to quickly rule out bounded reachability queries, it can also be useful in another way. Specifically, an element of the TPA sequence may turn out to be a *transition invariant* with respect to transition relation *Tr*.

Definition 1 (transition invariant). *We say that* $R(x, x')$ *is a* transition invariant *if* $Tr^* \subseteq R$*, i.e.,* $\forall x, x' \; Tr^*(x, x') \implies R(x, x')$*, where* Tr^* *is the reflexive transitive closure of* Tr*.*

Note that our definition is slightly simpler than that of [40], as it only depends on the transition relation and not, for example, on the initial states of the system.

If we find a transition invariant that does not relate any initial state with a bad state, we can immediately conclude that the system is safe. We show one way how to detect if a member of the TPA sequence is a transition invariant using SMT query.

Lemma 4. *Assume that for some* n*,* $ATr^{\leq n} \circ Tr \subseteq ATr^{\leq n}$ *or that* $Tr \circ ATr^{\leq n} \subseteq ATr^{\leq n}$*. Then* $ATr^{\leq n}$ *is a transition invariant.*

Proof. We consider the case $ATr^{\leq n} \circ Tr \subseteq ATr^{\leq n}$ and show that $Tr^* \subseteq ATr^{\leq n}$. The other case is analogous. Take any two states s, s' such that s' is reachable from s, i.e., $(s, s') \in Tr^*$. We show that $(s, s') \in ATr^{\leq n}$ by induction on d, the length of the path from s to s'. If $d \leq 2^n$ then $(s, s') \in ATr^{\leq n}$ by Eq. (1). Assume now that $d > 2^n$. Then there exists a state t such that t can be reached from s in $d - 1$ steps and $(t, s') \in Tr$. By induction, we have that $(s, t) \in ATr^{\leq n}$ and $(s, s') \in ATr^{\leq n} \circ Tr$. By our assumption it follows that $(s, s') \in ATr^{\leq n}$. □

Note that when a call to `IsReachable` on line 5 in Algorithm 1 returns ∅, the $n{+}1^{\text{st}}$ element of TPA sequence $ATr^{\leq n+1}$ does not relate any initial and bad state. Thus we can check at this point for the conditions of Lemma 4, and, if satisfied, we can immediately conclude that no counterexample (of any length) exists in the system and report safety.

In fact, to detect that no counterexample exists, the assumptions of Lemma 4 can be relaxed a bit. We can consider the restriction of these relations to only initial or bad states. The notation $A \lhd R$ denotes a domain restriction of a binary relation R to a set A, i.e., $(x, y) \in A \lhd R$ iff $(x, y) \in R \land x \in A$. Similarly $R \rhd B$ denotes the codomain restriction, i.e., $(x, y) \in R \rhd B$ iff $(x, y) \in R \land y \in B$.

Lemma 5. *Assume that for some* n $Init \lhd ATr^{\leq n} \circ Tr \subseteq Init \lhd ATr^{\leq n}$*. Then* $Init \lhd Tr^* \subseteq Init \lhd ATr^{\leq n}$*. Similarly, if* $Tr \circ ATr^{\leq n} \rhd Bad \subseteq ATr^{\leq n} \rhd Bad$*, then* $Tr^* \rhd Bad \subseteq ATr^{\leq n} \rhd Bad$*.*

Proof. Same as the proof of Lemma 4, with appropriate restrictions.

Lemma 5 represents a weaker form of Lemma 4: it has a weaker assumption and a weaker conclusion. Nevertheless, the conclusion is still strong enough to ensure that no counterexample exists and conclude safety.

5 Experiments

We have implemented our TPA-based procedure (Algorithm 1) in our new CHC solver Golem[7]. Golem is built on top of the interpolating SMT solver OpenSMT [26]. In our experiments we used version 2.2.0 of OpenSMT[8].

To gauge the feasibility of our algorithm we performed a set of experiments. All experiments were conducted on a machine with AMD EPYC 7452 32-core processor and 8x32 GiB of memory. We compared our approach to the current state-of-the-art tools Eldarica 2.0.6 [25], IC3-IA 20.04.1 [15] and Z3 4.8.12 [39] (using both its BMC [9] and Spacer [30] engines), which were the top competitors in CHC-COMP 2020 and 2021 [43,21]. We used both versions of our algorithm in the experiments: using MBP (TPA-MBP) and QE (TPA-QE). The format of all the benchmarks is that of the constrained Horn clauses (CHCs) used in the CHC-COMP. Since IC3-IA's input format differs, all CHC benchmarks were translated to VMT format using the automated tool packaged with IC3-IA.[9]

The goal of the first experiment was to investigate the scalability of our algorithm with respect to the length of the counterexample and compare its performance to the state-of-the-art tools. We used the parametrized transition system from our motivating example in Section 3. The counterexample in this system has length $2N$ and we ran the tools on instances for N ranging from 1 to 511. The timeout was set to 300 seconds. Figure 2 shows the runtime of the tools for the given value of N.

TPA-MBP was able to report all instances as unsafe, needing *less than two seconds* for each instance. Eldarica, IC3-IA and Z3-BMC exhibit relatively stable pattern where the performance decreases rapidly with increasing N. Z3-Spacer, on the other hand, exhibits a curious behaviour where it is able to solve most of the instances (even though it is slower than TPA-MBP by at least an order of magnitude), but on a relatively large number of instances it times out, and we were not able to understand the pattern on which instances this happens. Quick look at the instances for $N < 100$ suggests that on some instances its behaviour is much closer to that of IC3-IA. Finally, TPA-QE also shows an interesting pattern in its runtime where its performance drops considerably on every power of two, and then it slowly improves for larger N until the next power of two.

This first experiment showed very promising results for TPA-MBP which benefited from the fact that the reason why shorter counterexamples do not exist can be summarized relatively easily. It scaled exceptionally well compared to the state-of-the-art tools, as well as TPA-QE.

To confirm the results from the first experiment, we continued with the second set of benchmarks representing instances of our targeted type of problems. They represent assertions over *multi-phase* loops, which are known to be difficult to analyze by state-of-the-art techniques. We took 54 safe multi-phase benchmarks

[7] https://github.com/usi-verification-and-security/golem; commit 4ea1a53

[8] https://github.com/usi-verification-and-security/opensmt

[9] Full results of the experiments available at http://verify.inf.usi.ch/horn-clauses/tpa/experiments. Artifact available at https://doi.org/10.5281/zenodo.5815911

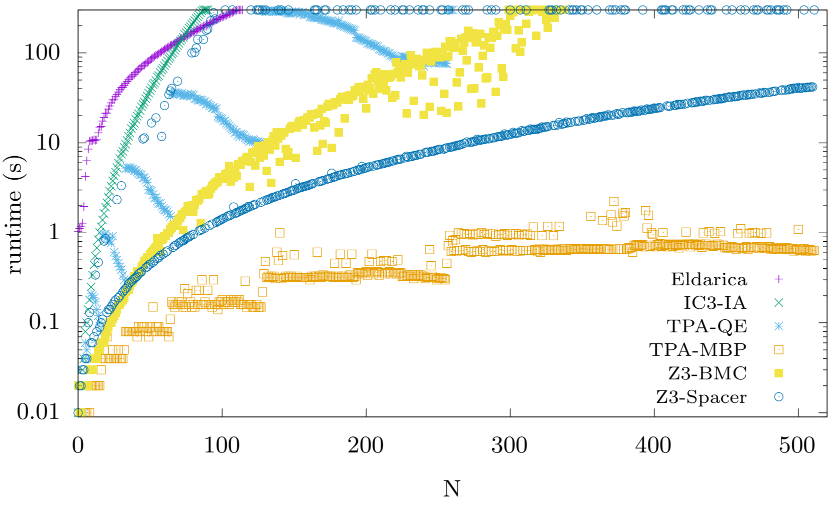


Fig. 2: Runtime for motivating example for N from 1 to 511 (log y-axis)

from CHC-COMP repository[10] and then for each benchmark created its unsafe version with a minor modification of the safety property.[11] In most cases this was done by negating one of the conjuncts of the property. In a few cases this resulted in a simple benchmark with a very short CEX (< 10 steps), but in most cases, the minimal counterexample is much larger, ranging from a few hundreds to a few tens of thousands of steps. There are even a few extremes where the minimal counterexample requires hundreds of thousands or even millions of steps.

With the timeout of 300 seconds, out of 54 benchmarks, TPA-QE solved 20 and TPA-MBP solved 35 benchmarks, beating the other tools among which Z3-Spacer performed the best, solving 20 benchmarks. The results are summarized in Figure 3 where the number of solved benchmarks by each tool is plotted against the time needed for their solving.

Overall, our tool solved 15 benchmarks that none of the other tools was able to solve and in general could be one or two orders of magnitude faster. There were two noticeable exceptions: benchmark 24 was uniquely solved by Z3 and benchmark 39 was uniquely solved by IC3-IA (for benchmark numbering, see the link in footnote 11). We found out that in the latter case our tool suffered from incompleteness in the decision procedure of OpenSMT for integer arithmetic, while in the former case the interpolation used by our algorithm was not producing good abstractions and we suffered from the need for frequent refinements.

We also examined the solved benchmarks for the length of the minimal counterexample they admit. The results are in line with the observations from our first experiments: Other tools could only solve benchmarks with a counterexample

[10] https://github.com/chc-comp/aeval-benchmarks

[11] Benchmarks available at https://github.com/blishko/chc-benchmarks.

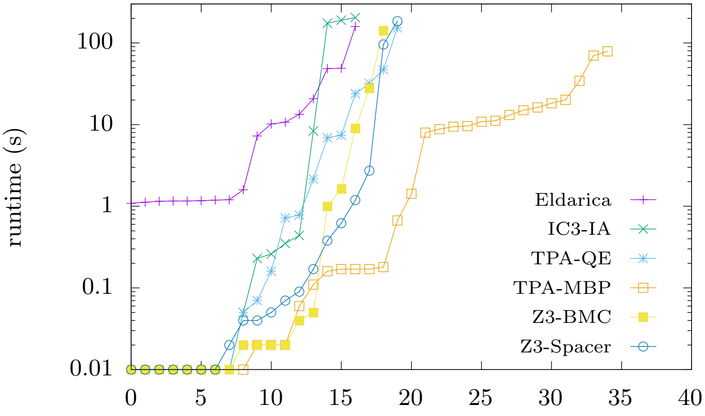


Fig. 3: Results on 54 multi-phase unsafe benchmarks

of up to a thousand steps (1001 steps in benchmark 17 solved by Z3-Spacer). TPA-QE matched this performance (1001 steps in benchmark 27), but TPA-MBP managed to solve benchmarks with a counterexample of more than *ten thousand* steps (17650 in benchmark 42). Thus, our technique significantly improves upon state-of-the-art with respect to the length of the counterexample it can detect.

Finally, we successfully tested our implementation on the safe version of the 54 multi-phase benchmarks and on the general set of 498 benchmarks from CHC-COMP'21, the category of transition systems over linear real arithmetic. TPA-MBP managed to prove 10 of the multi-phase benchmarks safe. Z3-Spacer, IC3-IA and Eldarica proved safe 9, 20 and 26 of these benchmarks, respectively. On the CHC-COMP LRA-TS benchmark set, TPA-MBP was able to solve 70 unsafe benchmarks (from 90+ known unsafe benchmarks in the set) and 67 safe benchmarks.

6 Related work

Loop acceleration [5,12,22] is a related approach for loop analysis that enables both proving safety and detection of deep bugs. It transforms the loop to a single quantifier-free formula representing all possible executions of the loop. While offering significant improvement for a limited types of integer loops, it is not applicable for code with control-flow divergence and/or data structures. Acceleration has also been combined with interpolation-based model-checking [13,24]. In contrast, our technique does not accelerate paths but builds over-approximations of bounded number of iterations. It is not restricted to any specific type of loops, and it works over any theory supporting interpolation and quantifier elimination.

Another technique for fast detection of deep counterexamples for C programs was proposed in [32]. Given a path through a loop, it computes a new path that *under-approximates* an arbitrary number of iterations of the original path. In contrast to loop acceleration, this technique only under-approximates the loop behaviour, but it can handle conditionals and richer background theories. Our approach targets the same goal but it is *over-approximating*, which allows for detecting (transition) invariants and proving safety. Their prototype aims at C programs only (and does not seem to be maintained anymore). Our implementation works on transition systems in the form of constrained Horn clauses (CHC) and thus is agnostic to the programming language.

Abstracting transition relation using interpolation has been employed in [27]. They use interpolation to compute and refine abstract version of the transition relation. However, they abstract only a single step of the transition relation. Instead, we use interpolation to compute relations that over-approximate multiple (and increasingly larger number of) steps of the transition relation.

Transition invariants [40] have been successfully employed for proving liveness properties, especially termination [33,41]. Our technique can discover transition invariants and use them to prove safety. However, in this paper we focused on finding counterexamples and the directed search for invariants is left for future work.

Our technique can find a possible application in automating test-case generation. A given program can be automatically annotated with assertions representing the reachability of all the branches. Having the goal to detect a set of input values for maximizing the test coverage [46], our technique would be called repeatedly to find many counterexamples for a subset of assertions (including deep ones) and prove the unreachability of the remaining ones.

7 Conclusion and Future Work

This paper introduces a novel model-checking algorithm for safety properties of transition systems with a focus on finding deep counterexamples. The idea is based on maintaining a sequence of transition formulas, called the *transition power abstraction* (TPA) sequence, where each element over-approximates a sequence of transition steps *twice as long as* its predecessor. The sequence is used in answering bounded reachability queries, which in turn results in new information that further refines the sequence. We proved the correctness of this algorithm and showed that it eventually finds a counterexample if one exists, assuming the background theory admits interpolation and quantifier elimination. For performance reasons, our implementation applies quantifier elimination lazily using model-based projection that lets the approach to outperform state-of-the-art on a class of problems with multi-phase loops. The experiments confirmed that it is able to detect counterexamples of much greater depth than existing tools within the same time constraints.

As future work, we plan to investigate possible improvements of the algorithm and tailor it for finding transition invariants. This would contribute to its ability to prove programs safety and enable the modular reasoning to support arbitrary systems of constrained Horn clauses.

References

1. Alt, L., Asadi, S., Chockler, H., Even Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Hifrog: SMT-based function summarization for software verification. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 207–213. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
2. Alt, L., Hyvärinen, A.E.J., Sharygina, N.: LRA interpolants from no man's land. In: Strichman, O., Tzoref-Brill, R. (eds.) HVC 2017. LNCS, vol. 10629, pp. 195–210. Springer, Cham (2017)
3. Asadi, S., Blicha, M., Fedyukovich, G., Hyv\"arinen, A., Even-Mendoza, K., Sharygina, N., Chockler, H.: Function summarization modulo theories. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 57, pp. 56–75. EasyChair (2018)
4. Asadi, S., Blicha, M., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N.: Incremental verification by SMT-based summary repair. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 77–82. IEEE (2020)
5. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: Acceleration from theory to practice. International Journal on Software Tools for Technology Transfer **10**(5), 401–424 (2008)
6. 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 http://smtlib.cs.uiowa.edu
7. Barrett, C., de Moura, L., Ranise, S., Stump, A., Tinelli, C.: The SMT-LIB initiative and the rise of SMT. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) Hardware and Software: Verification and Testing. pp. 3–3. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
8. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. Journal of Automated Reasoning **60**(3), 299–335 (Mar 2018)
9. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Tools and Alg. for the Const. and Anal. of Systems (TACAS '99). LNCS, vol. 1579, pp. 193–207 (1999)
10. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
11. Blicha, M., Hyvärinen, A.E.J., Kofroň, J., Sharygina, N.: Decomposing Farkas interpolants. In: Vojnar, T., Zhang, L. (eds.) Proc. TACAS 2019. LNCS, vol. 11427, pp. 3–20. Springer (2019)
12. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 227–242. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
13. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 428–442. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
14. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 46–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
16. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Logic **12**(1), 7:1–7:54 (Nov 2010)
17. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
18. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic **22**(3), 269–285 (1957)
19. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer (2010)
20. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: TACAS, Part I. LNCS, vol. 10805, pp. 251–269. Springer (2018)
21. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021)
22. Frohn, F.: A calculus for modular loop acceleration. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 58–76. Springer International Publishing, Cham (2020)
23. Govind, H., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–9 (2020)
24. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis. pp. 187–202. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
25. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: FMCAD. pp. 158–164. IEEE (2018)
26. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016)
27. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification. pp. 39–51. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
28. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Piskac, R., Talupur, M. (eds.) Proc. FMCAD 2016. pp. 85–92. IEEE (2016)
29. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 89–96 (2015)
30. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (Jun 2016)
31. Krajíček, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. The Journal of Symbolic Logic **62**(2), 457–486 (1997)
32. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods in System Design **47**(1), 75–92 (2015)

33. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 89–103. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
34. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2013. pp. 1–13. Springer, Heidelberg (2003)
35. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
36. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science **345**(1), 101–121 (2005)
37. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification (CAV '06). LNCS, vol. 4144, pp. 123–136 (2006)
38. McMillan, K.L.: Lazy annotation revisited. In: Proc. CAV 2014. LNCS, vol. 8559, pp. 243–259. Springer (2014)
39. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. pp. 337–340. Springer, Heidelberg (2008)
40. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004. pp. 32–41 (2004)
41. Podelski, A., Rybalchenko, A.: Transition invariants and transition predicate abstraction for program termination. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 3–10. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
42. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic **62**(3), 981–998 (1997)
43. Rümmer, P.: Competition report: CHC-COMP-20. Electronic Proceedings in Theoretical Computer Science **320**, 197–219 (Aug 2020)
44. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 703–719. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
45. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD 2014. pp. 1–8. IEEE (2009)
46. Zlatkin, I., Fedyukovich, G.: Maximizing branch coverage with constrained horn clauses. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg (2022)

Searching for Ribbon-Shaped Paths in Fair Transition Systems

Marco Bozzano, Alessandro Cimatti, Stefano Tonetta, Viktoria Vozarova(✉)

Fondazione Bruno Kessler (FBK)
via Sommarive, 18
Trento 38123, Italy
{bozzano,cimatti,tonettas,vvozarova}@fbk.eu

Abstract. Diagnosability is a fundamental problem of partial observable systems in safety-critical design. Diagnosability verification checks if the observable part of system is sufficient to detect some faults. A counterexample to diagnosability may consist of infinitely many indistinguishable traces that differ in the occurrence of the fault. When the system under analysis is modeled as a Büchi automaton or finite-state Fair Transition System, this problem reduces to look for ribbon-shaped paths, i.e., fair paths with a loop in the middle.

In this paper, we propose to solve the problem by extending the liveness-to-safety approach to look for lasso-shaped paths. The algorithm can be applied to various diagnosability conditions in a uniform way by changing the conditions on the loops. We implemented and evaluated the approach on various diagnosability benchmarks.

Keywords: Diagnosability· Model checking · Liveness to safety

1 Introduction

The design of fault detection mechanisms is a standard part of the design of safety-critical systems. Faults are usually not directly observable. They are diagnosed by observing a sequence of observations and inferring the value of unobservable variables based on a system model. A fundamental question for the design of such partially observable systems is to determine if it always possible to detect a fault. Diagnosability verification is the problem of checking whether the available sensors are sufficient to determine the occurrence of a fault.

Historically, diagnosability verification is reduced to a model checking problem looking for a critical pair of indistinguishable traces that differ with respect to the fault. This pair witnesses the impossibility to detect the fault along such sequence of observations.

When considering fair transition systems, critical pairs are not sufficient and it is necessary to look for infinitely many indistinguishable traces. In case of finite state systems, such set of infinite traces can be represented by ribbon-shaped paths, i.e., paths with a loop in the middle. Previous solutions, hinted

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 543–560, 2022.
https://doi.org/10.1007/978-3-030-99524-9_30

in [16], were based on either bounded model checking, so not able to prove diagnosability (absence of the critical ribbon-shaped paths) or BDD-based fixpoint computation, which suffers from the problem of precomputing the fair states.

In this paper, we propose a new approach based on the liveness-to-safety construction [3], where the search for a (single) lasso shaped path is reduced to an invariant property. Like in liveness-to-safety, we use additional variables to guess the loopback states, which in the case of ribbon-shaped paths are used twice, the first time for the loop in the middle, the second time for the final lasso. Additional constraints are added to encode the looping conditions that must hold in the two loops for encoding the diagnosability problem. The algorithm can be applied to various diagnosability conditions in a uniform way by changing the conditions on the loops. We implemented and evaluated the approach on various diagnosability benchmarks. Different algorithms have tested to solve the resulting invariant model checking problem, showing better performance with respect to the fixpoint-based approach.

The main contribution of the paper is the extension of liveness-to-safety to generate an infinite number of traces. The set is in the form of a ribbon shape (in other words, in the form $a; b^*; c; d^\omega$) and may have applications beyond the diagnosability problem, e.g., to solve non-interference problems requiring infinitely many different traces [13] or to counterexample-guided abstraction refinement.

The rest of the paper is organized as follows. In Section 2, we give an overview of related work. Section 3 defines the necessary formal background. The main problem along with the original solution is presented in Section 4. Our main contribution is introduced in Section 5, where we present the novel solution and prove its correctness. Section 6 contains the experimental evaluation comparing our solution with the original one. Finally, in Section 7 we give conclusions and directions for future work.

2 Related Work

The problem of diagnosability [17] refers to the possibility of inferring some desired information (e.g., the occurrence of a fault) during the execution of a system, in a partially observable environment. Hence, diagnosability can be phrased using hyperproperties, namely as a property of the traces representing the execution of the system [5,16].

In [16] it has been shown that the problem of diagnosability under fairness can be reduced to the search for ribbon-shaped paths, i.e. paths with a loop in the middle, where specific conditions on the occurrence of faults are imposed. Historically, diagnosability has been defined in the context of Discrete-Event Systems [17], without taking fairness into account. In [14] fairness is considered only in the context of live systems, i.e. under the hypothesis that every finite trace can be extended to an infinite fair trace, and fair diagnosability is introduced only informally. In this context, our ribbon-shaped fair critical pair corresponds to the critical pair of [14], where the faulty trace must be fair while the nominal trace may be unfair.

A construction similar to ribbon-shaped paths, called *doubly pumped lasso*, is used in [13] as a building block to address the problem of model checking a class of quantitative hyperproperties, as in the problem of quantitative non-interference (i.e., bound the amount of information about some secret inputs that may be leaked through the observable outputs of the system).

In [13] the problem of verifying quantitative hyperproperties is addressed using a model checking algorithm based on model counting, which is shown to have a better complexity than using an HyperLTL model checker, and a Max#SAT-based implementation. In [16], the authors address the problem of checking diagnosability using an extension of the classical twin-plant construction [15] and an LTL model checker. The approach we use in this paper builds upon the approach of [16], but uses an extension of the liveness-to-safety approach [3], instead. The extension omits the computation of fair states and keeps the representation of the system symbolic, which is more space efficient. The problem is reduced to the reachability problem. The problem is well-studied, thus we may take advantage of already developed algorithms for checking reachability.

3 Background

3.1 Symbolic Fair Transition Systems

The plant under analysis is represented as a finite-state *symbolic fair transition system* (SFTS). An SFTS is a tuple $\langle V, I, T, F\rangle$, where V is a finite set of Boolean state variables; I is a formula over V defining the initial states, T is a formula over V, V' (with V' being the next version of the state variables) defining the transition relation, and F is a set of formulas over V defining the fairness conditions. If $F = \emptyset$, we call it a *symbolic transition system* (STS) and write $\langle V, I, T\rangle$.

We remark that the choice of representing the plant in form of an SFTS does not restrict the generality of the framework. In fact, it is possible to encode labeled transition systems and discrete event systems.

A *state* s is an assignment to the state variables V. We denote with s' the corresponding assignment to V'. Given an assignment to a set V of Boolean variables, we also represent the assignment as the set of variables that are assigned to true. Given a state s and a subset of variables U, we denote with $s_{|U}$ the restriction of s to the variables in U.

In the following we assume that an SFTS $P \doteq \langle V, I, T, F\rangle$ is given.

Given a sequence of states σ, we denote with σ^k the sequence obtained by repeating σ for k times, and σ^ω the sequence obtained by repeating σ for an infinite number of times.

Given a state s_0 of P, a *trace of P starting from s_0* is an infinite sequence $\pi \doteq s_0, s_1, s_2, \ldots$ of states starting from s_0 such that, for each $k \geq 0$, $\langle s_k, s_{k+1}\rangle$ satisfies T, and for all $f \in F$, for infinitely many $i \geq 0$, the formula f is true in

s_i. If s_0 is initial, i.e., it satisfies I, then we say that π is a *trace of* P. We write Π_P for the set of traces of P.

We denote with $\pi[k]$ the $k+1$-th state s_k of π. We say that s is *reachable* (in k steps) in P iff there exists a sequence $\pi = s_0 s_1 \dots s_k$, where $s_k = s$, s_0 satisfies I and every $\langle s_i, s_{i+1}\rangle$ satisfies T. A state s is fair if there exists a trace starting from s.

Given a trace $\pi \doteq s_0, s_1, s_2, \dots$ and a subset of variables $U \subseteq V$, we denote by $\pi_{|U} \doteq s_{0|U}, s_{1|U}, s_{2|U}, \dots$ the projection over the variables in U.

Let $S^1 = \langle V^1, I^1, T^1, F_1\rangle$ and $S^2 = \langle V^2, I^2, T^2, F_2\rangle$ be two SFTSs. We define a *synchronous product* $S^1 \times S^2$ as the SFTS $\langle V^1 \cup V^2, I^1 \wedge I^2, T^1 \wedge T^2, F_1 \cup F_2\rangle$. Every state s of $S^1 \times S^2$ is an assignment to the two sets of state variables V^1 and V^2 such that $s^1 = s_{|V^1}$ is a state of S^1 and $s^2 = s_{|V^2}$ is a state of S^2.

Let p be a propositional formula over V. We write $s \models p$ iff s satisfies p, and $\pi, i \models p$ if $\pi[i]$ satisfies p. We write $P \models p$ iff for all reachable s in P it holds that $s \models p$. Let φ be a formula over an infinite trace expressed in LTL [12]. We write $\pi \models \varphi$ iff φ is true on the trace π. We write $P \models \varphi$ iff for all traces π in Π_P it holds that $\pi \models \varphi$.

In the rest of the presentation, we sometimes use a context, which we express as an LTL formula Ψ, to restrict the set of traces of the plant. This is useful to address the problem of diagnosability under assumptions. Note that, since our framework supports plants with fairness constraints, the incorporation of the context can be done (see, e.g., [10]) by converting the context into an SFTS S_Ψ (representing the monitor automaton for the LTL formula) and replacing the plant P with $P \times S_\Psi$ (the synchronous product of the plant with the monitor automaton).

The Twin Plant Construction The twin plant construction of a plant P over a subset $Y \subseteq V$ of variables (the observable variables), denoted $\textsc{Twin}(P, Y)$ and originally proposed by [15], is based on two copies of P, such that a trace in the twin plant corresponds to a pair of traces of P. In the security domain, two copies of a system used for verification are known as a self-composition [2].

The twin plant can be defined as the synchronous product of two copies of the SFTS corresponding to the plant. Formally, given a plant $P = \langle V, I, T, F\rangle$, we denote with $P_L \doteq \langle V_L, I_L, T_L, F_L\rangle$ and $P_R \doteq \langle V_R, I_R, T_R, F_R\rangle$ the ('left' and 'right') copies of P, obtained by renaming each variable v as v_L or v_R, respectively (i.e., if $\square \in \{L, R\}$, then $V_\square$ stands for the set of variables $\{v_\square \mid v \in V\}$. Moreover, we define a formula $\textsc{ObsEq}$ stating that the sets of observable variables of the two copies are equal at the given point. The twin plant of P is defined as follows.

Definition 1 (Twin Plant). *Given a set of variables $Y \subseteq V$, the* twin plant *of $P = \langle V, I, T, F\rangle$ is the SFTS $\textsc{Twin}(P, Y) \doteq P_L \times P_R$. Moreover, we define the formula $\textsc{ObsEq} \doteq \bigwedge_{v \in Y} v_L = v_R$.*

There is a one-to-one correspondence between $\Pi_P \times \Pi_P$ (pairs of traces of P) and $\Pi_{\textsc{Twin}(P,Y)}$ (traces of $\textsc{Twin}(P, Y)$). A trace of $\textsc{Twin}(P, Y)$: $\pi \doteq (s_{0,L}, s_{0,R})$,

$(s_{1,L}, s_{1,R}), \ldots$ can be decomposed into two traces of P: $\mathit{Left}(\pi) \doteq s_{0,L}, s_{1,L}, \ldots$ and $\mathit{Right}(\pi) \doteq s_{0,R}, s_{1,R}, \ldots$. Conversely, given two traces π_L and π_R in Π_P, there is a corresponding trace in $\Pi_{\textsc{Twin}(P,Y)}$, denoted by $\pi_L \times \pi_R$.

3.2 Liveness to Safety (L2S).

The *liveness-to-safety reduction* (L2S) [3] is a technique for reducing an LTL model checking problem on a finite-state transition system to an invariant model checking problem. The idea is to encode the absence of a lasso-shaped path violating the LTL property $\mathbf{FG}\neg f$ as an invariant property.

The encoding is achieved by transforming the original transition system S to the transition system S_{L2S}, introducing a set $\overline{X}$ of variables containing a copy $\overline{x}$ for each state variable x of the original system, plus additional variables *seen*, *triggered* and *loop*. Let $S \doteq \langle X, I, T\rangle$. L2S transforms the transition system in $S_{\mathrm{L2S}} \doteq \langle X_{\mathrm{L2S}}, I_{\mathrm{L2S}}, T_{\mathrm{L2S}}\rangle$ so that $S \models \mathbf{FG}\neg f$ if and only if $S_{\mathrm{L2S}} \models \neg \mathit{bad}_{\mathrm{L2S}}$, where:

$$\begin{aligned}
X_{\mathrm{L2S}} &\doteq X \cup \overline{X} \cup \{\mathit{seen}, \mathit{triggered}, \mathit{loop}\}\\
I_{\mathrm{L2S}} &\doteq I \wedge \neg \mathit{seen} \wedge \neg \mathit{triggered} \wedge \neg \mathit{loop}\\
T_{\mathrm{L2S}} &\doteq T \wedge \big[\textstyle\bigwedge_X \overline{x} \iff \overline{x}'\big]\\
&\quad \wedge\big[\mathit{seen}' \iff (\mathit{seen} \vee \textstyle\bigwedge_X (x \iff \overline{x}))\big]\\
&\quad \wedge\big[\mathit{triggered}' \iff (\mathit{triggered} \vee (f \wedge \mathit{seen}'))\big]\\
&\quad \wedge\big[\mathit{loop}' \iff (\mathit{triggered}' \wedge \textstyle\bigwedge_X (x' \iff \overline{x}'))\big]\\
\mathit{bad}_{\mathrm{L2S}} &\doteq \mathit{loop}
\end{aligned}$$

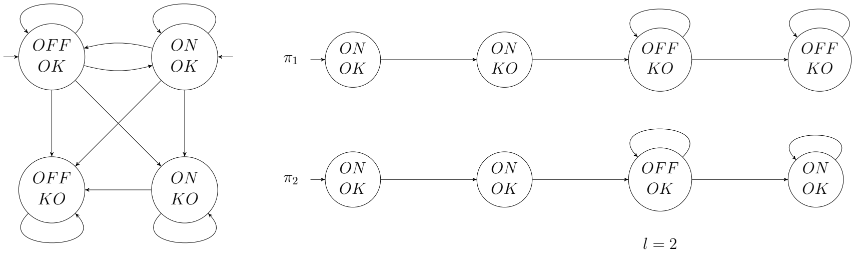
The variables $\overline{X}$ are used to non-deterministically guess a state of the system from which a reachable fair loop starts. The additional variables are used to remember that the guessed state was seen once and that the signal f was true at least once afterwards.

4 The Problem of Ribbon-Shaped Paths

4.1 The Diagnosability Problem

The *observable part* $\mathit{obs}(s)$ of a state s is the projection of s on the subset Y of observable state variables. Thus, $\mathit{obs}(s) \doteq s_{|Y}$. The observable part of π is $\mathit{obs}(\pi) \doteq \mathit{obs}(s_0), \mathit{obs}(s_1), \mathit{obs}(s_2), \ldots = \pi_{|Y}$. Given two traces π_1 and π_2, we denote by $\textsc{ObsEqUpTo}(\pi_1, \pi_2, k)$ the condition saying that, for all i, $0 \le i \le k$, $\mathit{obs}(\pi_1[i]) = \mathit{obs}(\pi_2[i])$.

Let β be a formula over V representing the fault condition to be diagnosed. We call β a diagnosis condition. A system is diagnosable for β if there exists a bound d such that after the occurrence of β, an observer can infer within d steps that β indeed occurred. This means that any other trace with the same observable part contains β as well. Formally, it was first defined in [17] as follows.



(a) light bulb (b) RCP

Fig. 1: The light bulb example and an example of a ribbon-shaped critical pair in the light bulb.

Definition 2 (Diagnosability). *Let P be a plant and β a diagnosis condition. P is* diagnosable *for β iff there exists $d \geq 0$ such that for every trace π_1 and index $i \geq 0$ such that $\pi_1, i \models \beta$, it holds:*

$$(\exists j \in \mathbb{N}\ i \leq j \leq i+d \cdot (\forall \pi_2 \cdot \textsc{ObsEqUpTo}(\pi_1, \pi_2, j) \Rightarrow \exists k \in \mathbb{N}\ k \leq j \cdot \pi_2, k \models \beta)).$$

The above definition requires a global bound, while when considering fair transition systems it is possible that the occurrence of β can be inferred eventually, but without a fixed bound. That is the motivation of extending the definition to fair diagnosability [16].

Definition 3 (Fair Diagnosability). *Let P be a plant and β a diagnosis condition. P is* fair-diagnosable *for β iff for every trace π_1, there exists $d \geq 0$ such that for every index $i \geq 0$ such that $\pi_1, i \models \beta$, it holds:*

$$(\exists j \in \mathbb{N}\ i \leq j \leq i+d \cdot (\forall \pi_2 \cdot \textsc{ObsEqUpTo}(\pi_1, \pi_2, j) \Rightarrow \exists k \in \mathbb{N}\ k \leq j \cdot \pi_2, k \models \beta)).$$

Example 1. Consider the state machine of a light bulb as shown in Figure 1a, with the observable value OFF/ON and the diagnosis condition $\beta \doteq KO$. Consider the following context: $\mathsf{G}(KO \rightarrow \mathsf{F}\ OFF) \wedge \mathsf{G}(OK \rightarrow \mathsf{F}\ ON)$. Intuitively, the LTL formula states that globally a state where KO holds is followed eventually by a state where OFF holds, and similarly a state where OK holds is followed eventually by a state where ON holds. Therefore, if the execution reaches KO, it will eventually go into state OFF/KO and remain there forever. If an execution is instead always OK, then it will visit infinitely often the state ON/OK. We can prove that condition β is not fair-diagnosable according to Def. 3. In fact, for every j, there exists a trace without β that is observationally equivalent up to j to the trace with β. Notice how the fairness condition causes the observations after a failure to always diverge *eventually*, but that this event can be delayed indefinitely.

4.2 Ribbon-Shaped Critical Pairs

Figure 2 illustrates the concept of ribbon-shaped paths. The formal definition is as follows.

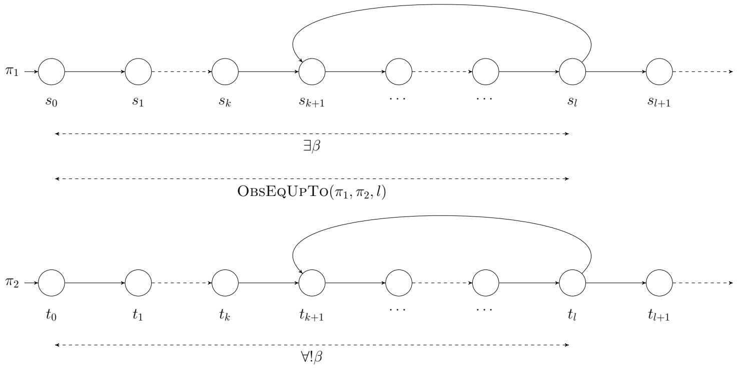


Fig. 2: Ribbon-shaped critical pair

Definition 4 (Ribbon-Shaped Critical Pairs (RCP)). *Let P be a plant and β a diagnosis condition. We say that $\pi_1, \pi_2 \in \Pi_P$ are a ribbon-shaped critical pair for the diagnosability of β iff there exist k, l such that $0 \leq k \leq l$ and:*

1. $\pi_1[l] = \pi_1[k]$ *and* $\pi_2[l] = \pi_2[k]$*;*
2. ObsEqUpTo(π_1, π_2, l)*;*
3. $\pi_1, i \models \beta$ *for some* $i, 0 \leq i \leq l$*;*
4. $\pi_2, i \not\models \beta$ *for all* $i, 0 \leq i \leq l$*.*

For fair diagnosability, the definition is similar:

Definition 5 (Ribbon-Shaped Fair Critical Pairs (RFCP)). *Let P be a plant and β a diagnosis condition. We say that $\pi_1, \pi_2 \in \Pi_P$ are a ribbon-shaped fair critical pair for the diagnosability of β iff there exist k, l such that $0 \leq k \leq l$ and:*

1. $\pi_1[l] = \pi_1[k]$ *and* $\pi_2[l] = \pi_2[k]$*;*
2. π_1 *is in the form* $s_0, s_1, \dots s_k, (s_{k+1}, \dots s_l)^\omega$*;*
3. ObsEqUpTo(π_1, π_2, l)*;*
4. $\pi_1, i \models \beta$ *for some* $i \geq 0$*;*
5. $\pi_2, i \not\models \beta$ *for all* $i, 0 \leq i \leq l$*.*

In this paper, we use a slightly different definition than the one given in [16]. Definition 5 includes an additional constraint on π_1 by requiring a loop shape. However, these two definitions are equivalent and the proof of it can be found in the extended version of the paper.

Example 2. Fig. 1b shows an example of a ribbon-shaped critical pair for the light bulb of Example 1.

We can prove that, in the general case, β is not diagnosable if and only if there exists a ribbon-shaped critical pair. In other words, ribbon-shaped critical pairs are necessary and sufficient for diagnosability violation. The following theorem is adapted and extended from [16] and can be proved in a similar way.

Theorem 1 (RCP necessary and sufficient for diagnosability). *Let P be a plant. P is not diagnosable for β iff there exists a ribbon-shaped critical pair for the diagnosability of β. P is not fair diagnosable for β iff there exists a ribbon-shaped fair critical pair for the fair diagnosability of β.*

The proof can be done similarly as in [16]. The theorem in [16] is proved for asynchronous systems while here we assume that the plants in the twin plant are synchronized on the observable part.

4.3 Fixpoint-based Algorithm

The ribbon-shaped structure requires to eventually reach a loop (the ribbon), from which it is possible to branch with a fair suffix (the final lasso). Therefore, it combines path and branching conditions, and can be encoded into a CTL* formula [16] over variables of the twin plant. We can verify whether the formula holds in the twin plant using a fixpoint-based algorithm. Actually, the specific structure allows for a simple implementation on top of standard BDD-based model checking [16]: it is sufficient first to compute the set of fair states, then to compute the set of fair states staying forever in the looping condition, and finally to look for an initial state reaching such loop.

The main issue of this approach is the computation of fair states, which is performed independently from the diagnosis condition and may be a bottleneck in case of complex fairness conditions.

5 Extended Liveness to Safety

In this section, we propose a novel algorithm for finding RCPs and RFCPs in fair symbolic transition systems. The algorithm extends L2S such that it searches for two consequent loops instead of one. We define a ribbon structure, which is constructed from the twin plant. The ribbon structure is parametrized, thus it can be used for finding both RCPs and RFCPs with only a slight modification. We prove that a certain state is reachable in the ribbon structure if and only if there exists an RCP/RFCP in the original structure.

The ribbon structure extends the twin plant of the original structure with a new copy of state variables, new flags, and new transitions that constrain the behaviour of the new variables and the flags. In the following, we describe how the twin plant is extended and we formally define the ribbon structure.

5.1 Definition of the L2S Extension

The ribbon structure is parametrized by SFTS P, two propositional formulas p and q and two sets of propositional fairness conditions F_1 and F_2. These parameters are later instantiated depending on the specific ribbon-shaped path that is considered. In particular, p represents the diagnosis condition β in the left copy of the twin plant and q represents the negation of β in the right copy conjoined with the constraint to force the same observations on the two copies.

The ribbon structure $P_\sim$ and the propositional formula $\varphi_\sim$ are defined such that any path ρ of $P_\sim$ on which $\varphi_\sim$ is reached satisfies the following conditions:

- ρ contains two consequent loops L_1 and L_2 that satisfy fairness conditions F_1 and F_2 respectively;
- p is satisfied in some state of ρ before the end of the first loop;
- q is satisfied in all states of ρ before the end of the first loop.

In the rest of this section, we formally define the set of variables, the initial formula and the transition formula using the parameters described above.

Variables Similarly as in the original liveness-to-safety reduction, we create a copy of all state variables of the twin plant. The copy variables serve as a guess of the state representing a loopback. The variables are denoted by overline and defined as $\overline{V} = \{\overline{v} \mid v \in V\}$, where V is the set of variables of the twin plant P. The variables are reused both for the first and the second loop, where the second loop is a fair loop.

The flags are auxiliary variables used to monitor whether a loop was found and whether all loop conditions were satisfied. The set of flags is defined as $V_\mathbf{m} = \{\mathbf{m}_{seen}, \mathbf{m}_{L_1}, \mathbf{m}_p, \mathbf{m}_q\} \cup \bigcup_{f_i \in F_1} \mathbf{m}_{1,i} \cup \bigcup_{f_i \in F_2} \mathbf{m}_{2,i}$. The intuition behind each flag is as follows:

$\mathbf{m}_{seen}$ is true $\iff$ the loopback (either the first or the second one) was already seen and is saved in $\overline{V}$;
$\mathbf{m}_{L_1}$ is true $\iff$ the first loop was already found;
$\mathbf{m}_p$ is true $\iff$ p was true;
$\mathbf{m}_q$ is true $\iff$ q was true in all previous states;
$\mathbf{m}_{1,i}$ is true $\iff$ $f_i \in F_1$ was true in the first loop;
$\mathbf{m}_{2,i}$ is true $\iff$ $f_i \in F_2$ was true in the second loop.

In addition, when $\mathbf{m}_{seen}$ is true, the current state is in a loop. If $\mathbf{m}_{L_1}$ is false, it is in the first loop. Otherwise, the first loop was already found and the current state is in the second loop.

Auxiliary Formula The following formula φ_{L_1} states requirements for finding the loopback of the first loop L_1. We need that the conditions on p and q are satisfied and that L_1 was yet not found. In addition, we need that the fairness conditions were true and that the current state is the same as the guessed loopback.

$$\varphi_{L_1} := \mathbf{m}_p \wedge \mathbf{m}_q \wedge \neg\mathbf{m}_{L_1} \wedge \mathbf{m}_{seen} \wedge \bigwedge_{f_i \in F_1} \mathbf{m}_{1,i} \wedge \bigwedge_{v \in V} v = \overline{v}$$

Initial Formula All flags besides $\mathbf{m}_q$ are initialized to false, $\mathbf{m}_q$ is initialized to true:

$$\neg\mathbf{m}_{seen} \wedge \neg\mathbf{m}_{L_1} \wedge \neg\mathbf{m}_p \wedge \mathbf{m}_q \wedge \bigwedge_{f_i \in F_1} \neg\mathbf{m}_{1,i} \wedge \bigwedge_{f_i \in F_2} \neg\mathbf{m}_{2,i} \tag{I1}$$

Transition Formulas We define transitions (T1)–(T8) to ensure the correct behaviour of the introduced variables such that the conditions mentioned above are satisfied. The transitions and their intuitive descriptions are as follows.

- Anytime $\mathbf{m}_{seen}$ is set to true, in the next state the copied variables are set to the state variables of the current state:

$$\neg\mathbf{m}_{seen} \wedge {\mathbf{m}_{seen}}' \implies \bigwedge_{v\in V} \overline{v}' = v \tag{T1}$$

- If $\mathbf{m}_{seen}$ is true, the values of the copy variables are preserved also in the next state:

$$\mathbf{m}_{seen} \implies \bigwedge_{v\in V} \overline{v}' = \overline{v} \tag{T2}$$

- The flags $\mathbf{m}_{x,i}$ can change to true only when $f_i \in F_x$ is true and the current state is in L_x:

$$\bigwedge_{f_i\in F_1} \big(\, ({\mathbf{m}_{1,i}}' = \mathbf{m}_{1,i}) \vee ({\mathbf{m}_{i,1}}' \wedge \mathbf{m}_{seen} \wedge \neg\mathbf{m}_{L_1} \wedge f_i) \, \big) \tag{T3}$$

$$\bigwedge_{f_i\in F_2} \big(\, ({\mathbf{m}_{2,i}}' = \mathbf{m}_{2,i}) \vee ({\mathbf{m}_{2,i}}' \wedge \mathbf{m}_{seen} \wedge \mathbf{m}_{L_1} \wedge f_i) \, \big) \tag{T4}$$

- $\mathbf{m}_{L_1}$ can change to true only when the first loop was found, as specified above by φ_{L_1}, and it forces $\mathbf{m}_{seen}$ to be set to false:

$$({\mathbf{m}_{L_1}}' = \mathbf{m}_{L_1}) \quad \vee \quad (\varphi_{L_1} \wedge \neg{\mathbf{m}_{seen}}' \wedge {\mathbf{m}_{L_1}}') \tag{T5}$$

- $\mathbf{m}_{seen}$ can change to false only when L_1 was just found:

$$\mathbf{m}_{seen} \implies ({\mathbf{m}_{seen}}' \quad \vee \quad (\neg{\mathbf{m}_{seen}}' \wedge \neg\mathbf{m}_{L_1} \wedge {\mathbf{m}_{L_1}}')) \tag{T6}$$

- $\mathbf{m}_p$ can change to true only when p is true:

$$({\mathbf{m}_p}' = \mathbf{m}_p) \vee (p \wedge {\mathbf{m}_p}') \tag{T7}$$

- Anytime q is false, $\mathbf{m}_q$ goes to false and stays false:

$$(\neg\mathbf{m}_q \vee \neg q) \implies \neg{\mathbf{m}_q}' \tag{T8}$$

Note that the transitions (T3), (T4), (T5) and (T7) imply that flags $\mathbf{m}_{x,i}$, $\mathbf{m}_{L_1}$, $\mathbf{m}_p$ can change their value from false to true only once and then they stay true. The transition (T8) implies that $\mathbf{m}_q$ can change its value from true to false only once and then it stays false. Finally, (T6) implies that $\mathbf{m}_{seen}$ is set to false exactly once, when L_1 is found, and thus set to true exactly twice, when a loopback of either L_1 or L_2 is guessed.

Ribbon Structure Putting together the variables and formulas defined above, we give the following definition of the ribbon structure.

Definition 6. *For the plant $P = \langle V, I, T, F\rangle$, the propositional formulas p, q and the sets of propositional formulas F_1, F_2 over V, let $\langle V_\sim, I_\sim, T_\sim\rangle$ be a symbolic transition system where:*

- $V_\sim = V \cup \overline{V} \cup V_{\mathbf{m}}$;
- $I_\sim = I \wedge (I1)$;
- $T_\sim = T \wedge (T1) \wedge (T2) \wedge (T3) \wedge (T4) \wedge (T5) \wedge (T6) \wedge (T7) \wedge (T8)$.

We call this STS a ribbon structure and denote it by Ribbon(P, p, q, F_1, F_2).

To finish the reduction, we define the reachability condition. Intuitively, the condition should express that the second loop was found. This means that the first loop was already found, all fairness conditions in F_2 were true and the current state is the same as the guessed loopback:

$$\varphi_\sim := \mathbf{m}_{L_1} \wedge \mathbf{m}_{seen} \wedge \bigwedge_{f_i \in F_2} \mathbf{m}_{2,i} \wedge \bigwedge_{v \in V} v = \overline{v}.$$

In the next section, we show how the reachability in a ribbon structure is used to find RCPs and RFCPs and we prove that our construction is correct.

5.2 Correctness

The ribbon structure and the reachability condition are defined such that any satisfiable trace contains two consecutive loops. The definitions of RCP and RFCP describe only the first loop. Not all critical pairs contain the second loop. However, using the following propositions, we claim that the existence of a critical pair implies existence of a critical pair with two loops, where the first loop is as in the original pair and the second loop is fair. This fact is necessary to prove that if P contains a critical pair, we can find a critical pair with two loops in the ribbon structure.

Proposition 1. *Let π be a trace of an SFTS P. Then, any prefix of π can be extended to a trace π_F that ends with a fair loop.*

Proposition 2. *Let $\pi_1 = s_1, s_2, s_3 \ldots$, $\pi_2 = t_1, t_2, t_3 \ldots$ be traces of SFTS P that end with a fair loop. Then, the path $(s_1, t_1), (s_2, t_2), (s_3, t_3) \ldots$ is a trace of $P \times P$ that ends with a fair loop.*

The first proposition is true because we consider only finite systems. In a finite system, any infinite fair suffix contains a state that is repeated infinitely many times. Thus, there must be two occurrences of the state in between which all fairness conditions are true at least once. The second proposition is true because we can unroll the fair loops of π_1 and π_2 until both of them are in loop and then we match the period of the new fair loop in $\pi_1 \times \pi_2$ by taking the least common multiple of periods of the fair loops.

Theorem 2. *Let P be a plant and $P_\sim = \textsc{Ribbon}(\textsc{Twin}(P, Y), p, q, F_1, F_2)$ is a ribbon structure where $p = \beta_L$, $q = \neg\beta_R \wedge \textsc{ObsEq}$, $F_1 = \emptyset$, $F_2 = F_L \cup F_R$. There exists a ribbon-shaped critical pair in P for the diagnosability of β iff $P_\sim \models \varphi_\sim$.*

Proof. Here, we sketch the proof of the theorem. The full proof is given in the extended version of the paper. We separately prove both directions of the equivalence:

$\Longrightarrow$ We have $\pi_1, \pi_2 \in \Pi_P$ satisfying Definition 4. We prove that there is a trace $\rho \in \Pi_{P_\sim}$ such that $\rho \models \varphi_\sim$. At first, we show what the trace looks like and then we prove it is a trace of $P_\sim$. Let $\pi_{1,F}$, $\pi_{2,F}$ be a critical pair with two loops, where the first loop is equal to the loop in π_1, π_2 and the second loop is fair. We construct the path ρ as symbolized in Figure 3. The main idea is to set $\rho_{|V}$ to $\pi_{1,F} \times \pi_{2,F}$. The existence of loop bounds k, l, k', l' follows from the definition of RCP. In the copy variables $\rho_{|\overline{V}}$, we keep $(\pi_{1,F} \times \pi_{2,F})[k]$ until the first loop is found and then we switch to $(\pi_{1,F} \times \pi_{2,F})[k']$. Flags $\mathbf{m}_{seen}$ and $\mathbf{m}_{L_1}$ are set accordingly to the bounds of the loops. Flags $\mathbf{m}_p$ and $\mathbf{m}_{2,i}$ are set to true after conditions β_L and f_i respectively were true. The existence of such states where the conditions are satisfied follows from the definition of RCP. Flag $\mathbf{m}_q$ is true until the first loop is found, because from the definition of RCP we know that $\neg\beta_R$ and $\textsc{ObsEq}$ are true.
The formal definition of ρ and the full proof that $\rho \in \Pi_{P_\sim}$ and $\rho \models \varphi_\sim$ is given in the appendix.

$\Longleftarrow$ We have $P_\sim \models \varphi_\sim$, thus there is $\rho \in \Pi_{P_\sim}$ such that $\rho \models \varphi_\sim$. Assume we have such ρ. We show how to construct π_1 and π_2 from ρ and then we prove that π_1, π_2 are an RCP for P and β. Let us set $\pi_1 = \rho_{|V_L}$ and $\pi_2 = \rho_{|V_R}$. Let the bounds k, l, k', l' of the loops in π_1 and π_2 be the indices:

- l' is such that $\rho, l' \models \varphi_{L_2}$;
- $k' < l'$ is the greatest index such that $\rho, k' \models \neg\mathbf{m}_{seen}$;
- $l < k' + 1$ such that $\rho, l \models \neg\mathbf{m}_{L_1} \wedge \mathbf{m}_{L_1}{}'$, from the construction we know there is only one such l;
- $k < l$ is the greatest index such that $\rho, k \models \neg\mathbf{m}_{seen}$.

In Appendix A, we finish the prove by showing that π_1 and π_2 are an RCP.

Theorem 3. *Let P be a plant and $P_\sim = \textsc{Ribbon}(\textsc{Twin}(P, Y), p, q, F_1, F_2)$ is a ribbon structure where $p = \beta_L$, $q = \neg\beta_R \wedge \textsc{ObsEq}$, $F_1 = F_L$, $F_2 = F_L \cup F_R$. There exists a ribbon-shaped critical pair in P for the fair diagnosability of β iff $P_\sim \models \varphi_\sim$.*

The proof is very similar to the previous one. The only difference is the necessity to verify the fairness of the first loop, which is done the same way as the fairness of the second loop and thus straightforward.

6 Experimental Evaluation

We compared the proposed technique based on L2S and the technique based on the computation of fixpoints using BDD proposed in [16] and briefly described

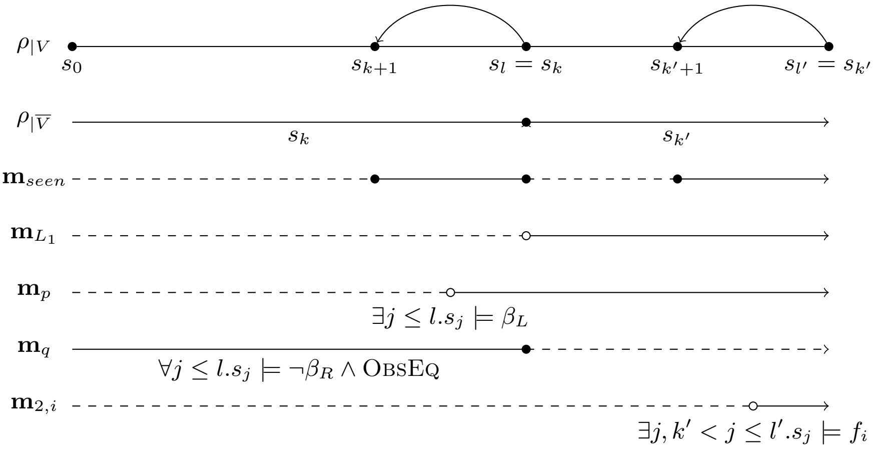


Fig. 3: The trace ρ as constructed in proof of Theorem 2. For each $\mathbf{m} \in V_{\mathbf{m}}$, a dashed line means $\rho, i \models \neg\mathbf{m}$, a full line and a full circle mean $\rho, i \models \mathbf{m}$, an empty circle means $\rho, i+1 \models \mathbf{m}$.

in Section 4.3. We implemented both algorithms in the xSAP platform [4] and tested them on benchmarks. The benchmarks, the tool and the scripts required to test it can be found online[1]. In this section, we at first introduce the implementation of the proposed technique and we describe the benchmarks. Then, we show comparison of the two techniques and we comment on their performance.

6.1 Implementation

We have implemented both the L2S algorithm and the BDD-based algorithm inside of the xSAP tool [4]. The algorithms make use of various procedures already implemented in nuXmv [8] and integrated in xSAP, mainly computation of fixpoint with BDDs [7] and different invariant model checking algorithms. The fair states are computed with the Emerson-Lei doubly-fixpoint algorithm [1]. The invariant model checking is implemented using engines based on standard verification algorithms IC3 [6], k-induction [18] and BDD-based fixpoint [11].

The input of each algorithm is a model in an SMV language[2], a list of observable variables of the model, a propositional diagnosis condition and an LTL formula representing the context. Both the model and the context are translated into Büchi automata and their parallel composition with the union of their accepting states is computed. The resulting set of accepting states is the set of fairness conditions. Then, a twin plant is constructed. The fixpoint-based algorithm is described in Section 4.

[1] http://es.fbk.eu/people/vvozarova/diag-rcp-search.zip

[2] see nuXmv manual htpps://nuxmv.fbk.eu)

Table 1: Properties of the used models.

model	#bool var	#reach	diam	#obs	#fairness
acex	31	$2^{19.4}$	96	5-21	1
autogen	99	$2^{12.0}$	20	4-20	1-4
cassini	176	$2^{44.2}$	8	5-58	1
guidance	98	$2^{47.5}$	70	5-62	1
pdist	83	$2^{11.0}$	31	5-41	1-4

In the L2S algorithm, we get the ribbon structure $P_\sim$ and the propositional formula $\varphi_\sim$ by extending the twin plant with new variables and transitions as defined in Section 6. The parameters p and q are constructed from the diagnosis condition and the set of observable variables. Finally, an arbitrary reachability algorithm is used to solve the reachability of $\varphi_\sim$ in the resulting system.

6.2 Benchmarks

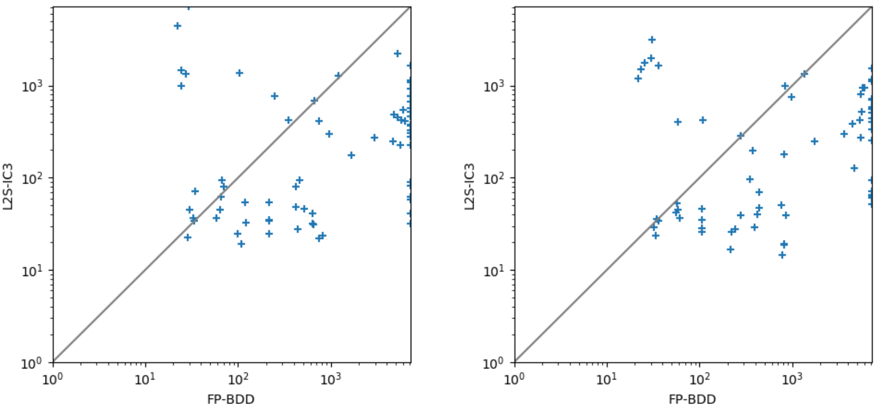
We selected several benchmarks modelling industrial use cases. The models are finite with boolean variables. For each model, we have specified a fault condition and possibly more sets of fairness conditions. Both the fault condition and the fairness conditions are given as propositional formulas. In Table 1, we give for each model the number of variables, the number of reachable states, the diameter of the state space, the sizes of sets of observable variables and the sizes of fairness condition sets.

Each benchmark was tested with more sets of observable variables and some were tested with more sets of fairness conditions. In sum, we have 72 examples for diagnosability and fair diagnosability problems and each instance was solved by BDD-based fixpoint approach and L2S approach with IC3, k-induction and BDD engines. This gives the total of 576 individual invocations of the xSAP tool. The experiments were run in parallel on a cluster with nodes with Intel Xeon CPU running at 2.27GHz with 8CPU, 48GB. The timeout for each run was two hours and the memory cap was set to 8GB.

6.3 Results

The results for selected examples are given in Table 2 and all results are plotted in Figure 4a for diagnosability and in Figure 4b for fair diagnosability. We compare the BDD-based fixpoint algorithm (FP-BDD) with L2S with IC3 engine algorithm (L2S-IC3). The k-induction engine was unable to prove diagnosability with the given bound on k (150), time and memory. In general, it performs better on cases where a counterexample exists, which are not of concern in this paper. The runs for L2S with BDD engine reached timeout in 127 out of 144 cases and for this reason we do not include it in the analysis.

As both figures and the table show, the approach using L2S extension is in most cases more effective than the BDD-based approach proposed in the previous



(a) diagnosability

(b) fair diagnosability

Fig. 4: Results for the diagnosability (a) and the fair diagnosability (b) comparing L2S approach with IC3 engine and BDD-based fixpoint computation approach. The axes represent time in seconds on a logarithmic scale.

literature. The novel technique manages to outperform the previous one in most cases, as is shown by the cases plotted below the diagonal line on each figure. Moreover, it manages to solve some cases in which the fixpoint-based algorithm timed out. For the acex model, FP-BDD performs better than L2S-IC3. This is because the model has few boolean variables, thus BDDs are smaller and operations on them are faster. In addition, IC3 needs 56-116 frames to prove non-reachability on acex, compared to 3-62 frames in other cases.

7 Conclusions and Future Work

In this paper, we considered the problem of proving the absence of a ribbon-shaped path, which is a core issue in proving diagnosability of fair transition systems. We conceived a new encoding extending the liveness-to-safety paradigm in order to search for two consecutive loops. We implemented the algorithm in the xSAP tool and evaluated it on various diagnosability benchmarks in comparison with a fixpoint-based solution.

The directions for future work are manifold: first, generalize the looping conditions to consider also problems different from diagnosability such as non-interference properties (as in [13]); second, exploit the generation of infinite sets of traces in counterexample-guided abstraction refinement, reducing the number of refinement iterations; finally, extend the approach to infinite-state systems, taking into account data variables that are updated in the loop (as in [9]).

Table 2: Results comparing L2S with IC3 engine and BDD-based algorithm. The times are given in seconds, TO stands for the timeout of 7200 seconds. All cases are diagnosable.

model	#obs	#fairness	diagnosability		fair diagnosability	
			L2S-IC3	FP-BDD	L2S-IC3	FP-BDD
acex	5	1	TO	**29.59**	3114.25	**30.58**
	9	1	4385.18	**22.25**	1493.26	**23.22**
	13	1	992.60	**24.25**	1203.87	**21.94**
	17	1	1450.14	**24.65**	1754.44	**25.35**
	21	1	1328.43	**27.22**	1996.66	**30.39**
autogen	4	1	676.89	**657.07**	**179.27**	809.45
		2	**300.14**	968.09	994.24	**840.69**
		3	**415.00**	741.83	**756.72**	988.33
		4	**228.62**	5638.46	**800.11**	5457.23
	16	1	**2231.98**	5188.65	**420.75**	5318.42
		2	**379.31**	TO	**586.94**	TO
		3	**411.57**	6300.83	**274.74**	5459.99
		4	**771.76**	TO	**574.25**	TO
	20	1	**482.92**	4741.88	**522.16**	5573.37
		2	**548.96**	6016.29	**943.53**	6043.12
		3	**426.54**	5728.60	**945.79**	5768.85
		4	**1134.01**	TO	**568.33**	TO
cassini	5	1	**31.85**	TO	**51.11**	TO
	10	1	**82.48**	TO	**60.35**	TO
	15	1	**90.62**	TO	**71.00**	TO
	20	1	**41.50**	TO	**61.95**	TO
	25	1	**62.39**	TO	**64.06**	TO
	58	1	**58.36**	TO	**64.65**	TO
guidance	5	1	425.76	**349.59**	**196.27**	370.38
	10	1	**173.75**	1663.83	**245.50**	1727.08
	15	1	**250.78**	4616.18	**128.19**	4678.15
	20	1	**271.89**	2928.52	**300.66**	3598.55
	25	1	**224.58**	TO	**507.58**	TO
	62	1	**278.82**	TO	**95.00**	TO
pdist	5	1	**95.19**	458.85	**96.81**	350.6
		2	**46.33**	511.33	**46.72**	435.57
		3	**48.09**	424.51	**40.19**	419.86
		4	**80.44**	420.94	**29.07**	388.84
	20	1	36.72	**32.96**	1635.52	**35.92**
		2	**22.47**	28.86	35.54	**34.19**
		3	71.29	**34.42**	**34.19**	35.74
		4	33.86	**33.65**	**28.98**	31.97
	25	1	773.29	**246.48**	285.85	**280.56**
		2	**54.20**	215.76	**38.83**	279.42
		3	**35.06**	216.55	**25.86**	219.13
		4	**24.75**	217.05	**16.75**	217.25
	41	1	**23.82**	818.28	**38.74**	859.25
		2	**31.33**	643.03	**50.58**	759.50
		3	**41.38**	633.24	**14.40**	782.73
		4	**22.21**	750.93	**18.93**	818.21

References

1. Allen Emerson, E., Lei, C.L.: Temporal reasoning under generalized fairness constraints. In: Monien, B., Vidal-Naquet, G. (eds.) STACS 86. pp. 21–36. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
2. BARTHE, G., D'ARGENIO, P.R., REZK, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011). https://doi.org/10.1017/S0960129511000193
3. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science **66**(2), 160–177 (2002). https://doi.org/https://doi.org/10.1016/S1571-0661(04)80410-9, https://www.sciencedirect.com/science/article/pii/S1571066104804109, fMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop)
4. Bittner, B., Bozzano, M., Cavada, R., Cimatti, A., Gario, M., Griggio, A., Mattarei, C., Micheli, A., Zampedri, G.: The xSAP Safety Analysis Platform. In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 533–539. Springer (2016)
5. Bozzano, M., Cimatti, A., Gario, M., Tonetta, S.: Formal Design of Asynchronous Fault Detection and Identification Components using Temporal Epistemic Logic. Logical Methods in Computer Science **11**(4), (2015). https://doi.org/10.2168/LMCS-11(4:4)2015, https://doi.org/10.2168/LMCS-11(4:4)2015
6. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)
7. Bryant, R.E.: Binary Decision Diagrams. In: Handbook of Model Checking, pp. 191–217. Springer (2018)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
9. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with Timed Transition Systems and Timed Temporal Properties. In: CAV (1). Lecture Notes in Computer Science, vol. 11561, pp. 376–386. Springer (2019)
10. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another Look at LTL Model Checking. Formal Methods in System Design **10**(1), 47–71 (1997)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
12. Emerson, E.: Temporal and Modal Logic. Handbook of theoretical computer science **2**, 995–1072 (1990)
13. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Chockler, H., Weissenbacher, G. (eds.) 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. 144–163. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_8, https://doi.org/10.1007/978-3-319-96145-3_8
14. Grastien, A.: Symbolic testing of diagnosability. In: International Workshop on Principles of Diagnosis (DX). pp. 131–138 (2009)
15. Jiang, S., Huang, Z., Chandra, V., Kumar, R.: A Polynomial-time Algorithm for Diagnosability of Discrete Event Systems. IEEE Transactions on Automatic Control **46**(8), 1318–1321 (2001)

16. M. Bozzano and A. Cimatti and S. Tonetta: Testing Diagnosability of Fair Discrete-Event Systems. In: Proc. International Workshop on Principles of Diagnosis (DX-19) (2019)
17. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Diagnosability of Discrete-event Systems. IEEE Transactions on Automatic Control **40**(9), 1555–1575 (1995)
18. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: FMCAD. Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer (2000)

CoVeriTeam: On-Demand Composition of Cooperative Verification Systems

Dirk Beyer and Sudeep Kanav

LMU Munich, Munich, Germany

Abstract. There is no silver bullet for software verification: Different techniques have different strengths. Thus, it is imperative to combine the strengths of verification tools via combinations and cooperation. CoVeriTeam is a language and tool for on-demand composition of cooperative approaches. It provides a systematic and modular way to combine existing tools (without changing them) in order to leverage their full potential. The idea of cooperative verification is that different tools help each other to achieve the goal of correctly solving verification tasks.
The language is based on verification artifacts (programs, specifications, witnesses) as basic objects and verification actors (verifiers, validators, testers) as basic operations. We define composition operators that make it possible to easily describe new compositions. Verification artifacts are the interface between the different verification actors. CoVeriTeam consists of a language for composition of verification actors, and its interpreter.
As a result of viewing tools as components, we can now create powerful verification engines that are beyond the possibilities of single tools, avoiding to develop certain components repeatedly. We illustrate the abilities of CoVeriTeam on a few case studies. We expect that CoVeriTeam will help verification researchers and practitioners to easily experiment with new tools, and assist them in rapid prototyping of tool combinations.

Keywords: Cooperative Verification · Tool Development · Software Verification · Automatic Verification · Verification Tools · Tool Composition · Tool Reuse

1 Introduction

As research in the field of formal verification advanced, the complexity of the programs under verification also kept on increasing. As a result, despite its successful application to the source code of large industrial and open-source projects [2, 3, 23, 27, 36], the current techniques fall short on solving many important verification tasks. It seems essential to combine the strengths of different verification techniques and tools to solve these tasks.

The verification community successfully applies different approaches to combine ideas: integrated approaches (source-code-based), where different pieces of source code are integrated into one tool [28], and off-the-shelf approaches (executable-based), where different executables from existing tools are combined

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 561–579, 2022.
https://doi.org/10.1007/978-3-030-99524-9_31

without changing them. The latter can be further classified into sequential and parallel portfolio [33], algorithm selection [37], and cooperative approaches [22].

The integrated approaches require development effort for adaptation or implementation of integrated components instead of building on existing mature implementations—the combination is very tight. On the other hand, the standard off-the-shelf approaches (portfolio [33] and selection [37]) let the tools run in isolation and the individual tools do not cooperate at all. The components do not benefit from the knowledge that is produced by other tools in the combination—the combination is very loose. In this work, we focus on cooperative verification, which is neither as tight as source-code integration nor as loose as portfolio and selection approaches—somewhere in between the two extremes.

Cooperative verification [22] is an approach to combine different tools for verification in such a way that they help each other solving a verification task, where the combinations are neither too tight nor too loose. Implementations include using a shared data base to exchange information (e.g., there are cooperative SAT solvers that use a shared set of learned clauses [34], and cooperative software verifiers that use a shared set of reached abstract states [14]) or pass information from one tool to the other (e.g., conditional model checkers [13, 25]). Cooperative verification aims to combine the individual strengths of these technologies to achieve better results. Our thesis is that *programming* (meta) verification systems based on *combination* and *cooperation* could be a promising solution. CoVeriTeam provides a framework to achieve this.

Developing such a tool is not straightforward. Various concerns that need to be addressed for developing a robust solution can be broadly divided in two categories: *concepts* and *execution.* (1) Concepts deal with defining the interfaces for tools, and with the mechanism for their combination. Before tools can cooperate, we need a common definition of tools based on their behavior. We need to categorize *what a tool does*, *what inputs it consumes*, and *what outputs it produces*, before we can use it in a cooperative framework with ease. In CoVeriTeam, we categorize tools in various types of verification *actors*, and the inputs and outputs produced by these actors in verification *artifacts*. The actors can be combined with the help of composition operators that define the mechanism of cooperation. (2) Execution is concerned with all issues during the execution of a tool. Actors first need to execute to cooperate. This opens another dimension of challenges and opportunities to improve the cooperation. To give two examples: a tool might have a too high resource consumption, thus, resources must be controlled and limited, and tools might interfere with other executing processes, thus, tools must be executed in isolated containers.

This paper presents CoVeriTeam, a language and tool for on-demand composition of cooperative verification systems that solves the above mentioned challenges. We contribute a domain-specific language and an execution engine. In the CoVeriTeam language, we can compose new actors based on existing ones using built-in composition operators. The existing components are not changed, but taken *off-the-shelf* from actor providers (technically: tool archives). We do not change existing software components: the composition is done *on-demand* (when needed by the user) and *on-the-fly* (it does not compile a new tool from the components). In other words, existing verification tools are viewed as off-the-shelf components, and can be used in a larger context to construct more powerful verification compositions. Our approach does not require writing code in programming languages used to develop the underlying components. In the CoVeriTeam language, the user can execute tools without fearing that they interact with the host system or with other tools in an unspecified way. The execution environment, as well as input and output, are controlled using the Linux features cgroups, name spaces, and overlay file systems. We use the BenchExec [20] system as library for convenient access to those OS features via a Python API.

Contributions. We make the following contributions:

1. a language to compose new verification tools based on existing ones,
2. an execution engine for on-the-fly execution of these compositions,
3. case studies implementing combinations in CoVeriTeam that were previously achieved only via hard-wired combinations, and
4. an open-source implementation and an artifact for reproduction.

In addition to the above mentioned contributions, CoVeriTeam provides the following features to the end user: (1) CoVeriTeam takes care of downloading and installing specified verification tools on the host system. (2) There is no need to learn command-line parameters of a verification tool because CoVeriTeam takes care of translating the input to the arguments for the underlying tool. This provides a uniform interface to a number of similar tools. (3) The off-the-shelf components (i.e., tools) are executed in a container, with resource limits, such that the execution cannot change (or even damage) the host system.

These features in turn enable a researcher or practitioner to easily experiment with new tools, and rapidly prototype new verification combinations. CoVeriTeam liberates the researcher who uses tool combinations from maintaining scripts that combine tools executions, and worrying about downloading, installing, and figuring out the command to execute a verification tool.

Impact. CoVeriTeam has already found use cases in the verification community: (1) It was used in a modular implementation of CEGAR [26] using off-the-shelf components [12]. (2) It was used for construction and evaluation of various verifier combinations [17]. (3) CoVeriTeam (wrapped in a service) was used in the software-verification competition 2021 and 2022 to help the participants debug issues with their tools (see Sect. 3 in [7]). Also, according to SV-COMP rules, a team is granted points only for those tasks whose result can be validated using a validator. Thus, a verifier-validator combination might be interesting for participants. With the help of CoVeriTeam such combinations can be easily constructed.

Also, the advent of many high-quality verifiers should lead to a certain level of standardization of the API and provided features. For example, tools for SMT or SAT solving are easy to use because of their standardized input language (e.g., SMTLIB for SMT solvers [4]). Consequently, such tools can be easily integrated into larger architectures as components. Our vision is that soon verifiers will be seen also as components that can be used in larger architectures just like SMT solvers are now integrated into verification tools.

Example 1 Witness Validation

Input: Program p, Specification s
Output: Verdict

```
verifier := Verifier(“Ultimate Automizer”)
validator := Validator(“CPAchecker”)
result := verifier.verify(p, s)
if result.verdict ∈ {True, False} then
    result = validator.validate (p, s, result.witness)
return (result.verdict, result.witness)
```

2 Running Example

We explain the idea of CoVeriTeam using a short example. Verifiers are complex software systems and might have bugs. Therefore, for more assurance a user might want to validate the result of a verifier based on the verification witness that the verifier produces [10]. Such a procedure is sketched in Example 1.

The user wanting to execute the procedure sketched in Example 1 would need to download the tools (verifier and validator), execute the verifier, check the result of the verifier, and then if needed connect the outputs of the verifier with the inputs of the validator. The user would quite possibly write a shell script to do this, which is cumbersome and difficult to maintain.

CoVeriTeam takes care of all the above issues. In the next section, we discuss the types, namely artifacts and actors, that are used in the CoVeriTeam language. After this, we explain the design and usage of the CoVeriTeam execution engine, and discuss the CoVeriTeam program for our validating verifier in Listing 1.

3 Design and Implementation of CoVeriTeam

We now explain details about the design and implementation of CoVeriTeam. First we discuss conceptual notions of actors, artifacts, and compositions; then we discuss execution concerns that a cooperative verification tool needs to handle. Then we delve deeper into implementation details where we discuss how an actor is created and executed. Last, we briefly explain the API that CoVeriTeam exposes and extensibility of this API.

3.1 Concepts

This section describes the language that we have designed for cooperative verification and on-demand composition. At first we describe the notion of artifacts and actors, and then the composition language to compose components to new actors.

Artifacts and Actors. Verification artifacts provide the means of information (and knowledge) exchange between the verification actors (tools). Figure 1 shows a hierarchy of artifacts, restricted to those that we have used in the case studies for evaluating our work. On a high level we divide verification artifacts in

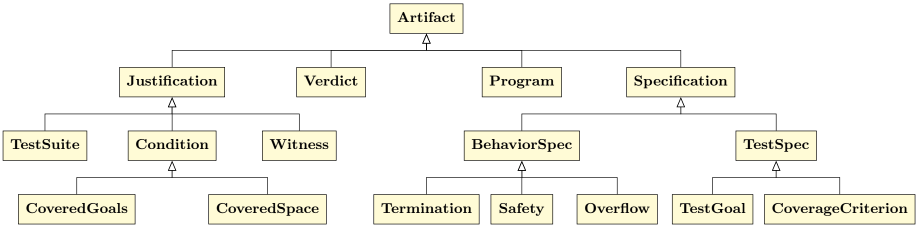


Fig. 1: Hierachy of Artifacts (arrows indicate an is-a relation)

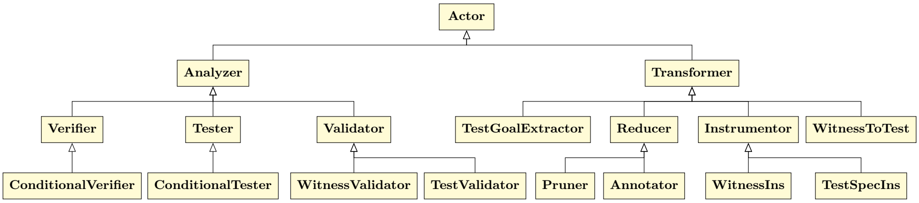


Fig. 2: Hierachy of Actors (arrows indicate an is-a relation)

the following kinds: *Programs*, *Specifications*, *Verdicts*, and *Justifications*. *Programs* are behavior models (can be further classified into programs, control-flow graphs, timed automata, etc.). *Specifications* include behavioral specifications (for formal verification) and test specifications (coverage criteria for test-case generation). *Verdicts* are produced by actors signifying the class of result obtained (True, False, Unknown, Timeout, Error). *Justifications* for the verdict are produced by an actor; they include test suites to justify an obtained coverage, or verification witnesses to justify a verification result.

Verification actors act on the artifacts and as a result either produce new artifacts or transform a given artifact for consumption by some other actor. Figure 2 shows a hierarchy of actors, restricted to those that we have used in the case studies for evaluating our work. We divide verification actors in the following types: *Analyzers* and *Transformers*. *Analyzers* create new knowledge, e.g., verifiers, validators, and test generators. *Transformers* instrument, refine, or abstract artifacts.

Composition. Actors can be composed to create new actors. Our language supports the following compositions: *sequence, parallel, if-then-else,* and *repeat.*

CoVeriTeam infers types and type-checks the compositions, and then either constructs a new actor or throws a type error. In the following, we use the notation I_a for the input parameter set of an actor a and O_a for the output parameter set of a. A parameter is a pair of name and *artifact type.* A *name clash* between two sets A and B exists if there is a name in A that is mapped to a different artifact type in B, more formally: $\exists (a, t_1) \in A,\ (a, t_2) \in B:\ t_1 \neq t_2$. The *actor type* is a mapping from input parameter set to output parameter set ($I_a \rightarrow O_a$).

Sequential. Given two actors $a1$ and $a2$, the sequential composition SEQUENCE ($a1$, $a2$) (Fig. 3a) constructs an actor that executes $a1$ and $a2$ in sequence, i.e., one after another. The composition is well-typed if there is no name clash between I_{a1} and $(I_{a2} \setminus O_{a1})$. This means that we allow same artifact to be passed to the second actor in sequence, but disallow the confusing scenario

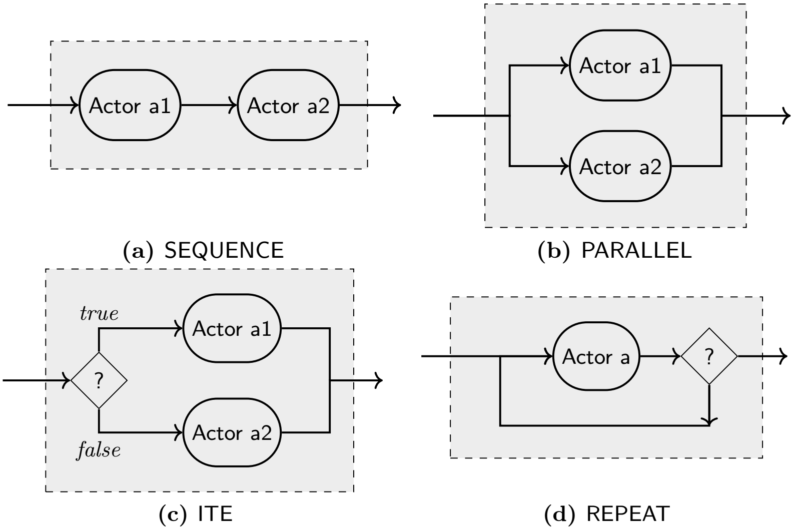


Fig. 3: Compositions in CoVeriTeam

where both actors expect an artifact with the same name but different type. The inferred type of the composition is $I_{a1} \cup (I_{a2} \setminus O_{a1}) \rightarrow O_{a2}$.

Parallel. Given two actors $a1$ and $a2$, the parallel composition PARALLEL ($a1$, $a2$) (Fig. 3b) constructs an actor that executes the actors $a1$ and $a2$ in parallel. The composition is well-typed if (a) there is no name clash between I_{a1} and I_{a2} and (b) the names of O_{a1} and O_{a2} are disjoint. The inferred type of the composition is $I_{a1} \cup I_{a2} \rightarrow O_{a1} \cup O_{a2}$.

ITE. Given a predicate *cond* and two actors $a1$ and $a2$, the *if-then-else* composition ITE (*cond*, $a1$, $a2$) (Fig. 3c) constructs an actor that executes the actor $a1$ if predicate *cond* evaluates to *true*, and the actor $a2$ otherwise. The composition is well-typed if (a) there is no name clash between *cond*, I_{a1}, and I_{a2}, and (b) the output parameters are the same ($O_{a1} = O_{a2}$). The inferred type of the composition is $I_{a1} \cup I_{a2} \cup \mathit{vars}(\mathit{cond}) \rightarrow O_{a1}$, where *vars* maps the variables used in a predicate to their artifact types. This allows us to define the condition *cond* using artifacts other than the inputs of I_{a1} and I_{a2}.

There are situations where $a2$ is not required and its explicit specification only increases complexity. So, we have relaxed the type checker and made $a2$ optional.

Repeat. Given a set *fp* and an actor a, the *repeat* composition REPEAT(*fp*, a) (Fig. 3d) constructs an actor that repeatedly executes actor a until a fixed-point of set *fp* is reached, that is, *fp* did not change in the last execution of a. The *repeat* composition feeds back the output of a from iteration n to a for iteration $n + 1$. Let us partition $I_a \cup O_a$ into three sets: $I_a \setminus O_a$, $O_a \setminus I_a$, and $I_a \cap O_a$. The parameters in $I_a \setminus O_a$ do not change their value and the parameters in $O_a \setminus I_a$ are accumulated if accumulatable, otherwise their value after the execution of the composition is the value from the last iteration. The composition is well-typed if $\mathit{fp} \subseteq \mathit{dom}(I_a \cap O_a)$, where *dom* returns the names of a parameter set. The inferred type of the composition is $I_a \rightarrow O_a$.

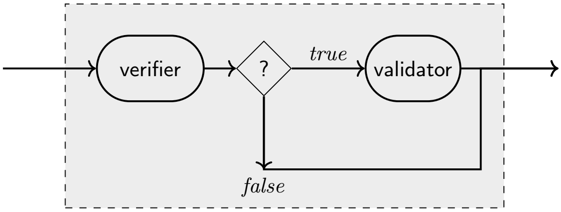


Fig. 4: CoVeriTeam implementation of the validating verifier from Example 1

Figure 4 shows the pictorial representation of our running example using these compositions. First a verifier is executed, then the validator is executed if the verifier returned True or False, otherwise (in case of Unknown, Timeout, Error) the validator is not executed and the output of the verifier is forwarded.

3.2 Execution Concerns

A tool for cooperative verification orchestrates the execution of verification tools. This means it needs to assemble the command for the tool, as well as handle the output produced by the tool. A verification tool might consume a lot of resources and a user might want to limit this. It might crash during execution, might interfere with other processes. CoVeriTeam needs to handle all these concerns.

Instead of developing our own infrastructure to handle these concerns, we reuse some of the features provided by BenchExec [20]: we use tool-info modules to assemble command lines and parse log output, RunExec (a component of BenchExec) to execute tools in a container and limit resource consumption.

Tool-Info modules are integration modules of the benchmarking framework BenchExec [20]. A typical tool-info module is a few lines of code used for assembling a command line and parsing the log output produced by the tool. It takes only a few hours to create one.[1] CoVeriTeam uses tool-info modules to pass artifacts to atomic actors (assemble command-line) and extract artifacts from the output produced by the atomic actor. Using tool-info modules gave us integration of more than 80 tools without effort, because such integration modules exist for most well-known verifiers, validators, and testers (as many researchers use BenchExec and provide such integration modules for their tools).

CoVeriTeam uses runexec to isolate tool execution to prevent interference with the execution environment and enforce resource limits. We also report back to the user the resources consumed by the tool execution as measured by runexec.

3.3 CoVeriTeam

Figure 5 provides an abstract view of the system. CoVeriTeam takes as input a program written in the CoVeriTeam language and artifacts. At first, the code generator converts this input program to Python code. This transformed

[1] We claim this based on our experience with tool developers creating their tool-info modules, which is a prerequisite for participating in SV-COMP or Test-Comp.

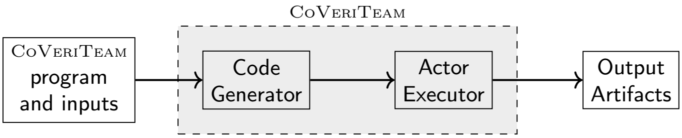


Fig. 5: Abstract view of the CoVeriTeam tool

```
1  verifier = ActorFactory.create(ProgramVerifier,
       "actors/uautomizer.yml");
2  validator = ActorFactory.create(ProgramValidator,
       "actors/cpa-validate-violation-witnesses.yml");
3
4  // Use validator if verdict is true or false
5  condition = ELEMENTOF(verdict, {TRUE, FALSE});
6  second_component = ITE(condition, validator);
7  // Verifier and second component to be executed in sequence
8  validating_verifier = SEQUENCE(verifier, second_component);
9
10 // Prepare example inputs
11 prog = ArtifactFactory.create(CProgram, prog_path);
12 spec = ArtifactFactory.create(BehaviorSpecification, spec_path);
13 inputs = {'program':prog, 'spec':spec};
14 // Execute the new component on the inputs
15 res = execute(validating_verifier, inputs);
16 print(res);
```

Listing 1: CoVeriTeam implementation of the validating verifier from Example 1

code uses the internal API of CoVeriTeam. Then this Python code is executed, which means the actor executor is called on the specified actor. This in turn produces output artifacts on successful execution of the actor.

There are four key parts of executing a CoVeriTeam program: creation of *atomic actors*, composition of *actors* (atomic or composite), creation of *artifacts*, and execution of the actors. We now give a brief explanation of these parts with the help of our running example. Listing 1 shows a CoVeriTeam implementation of the running example (Example 1).

Creation of an Atomic Actor. Atomic actors in CoVeriTeam provide an interface for external tools. CoVeriTeam uses the information provided in an actor-definition file to construct an atomic actor. Lines 1 and 2 in Listing 1 show the creation of atomic actors `verifier` and `validator` using the *ActorFactory* by providing the *ActorType* and the actor-definition file. Once constructed, this actor can be executed.

An *actor definition* is specified in a file in YAML format. It contains the information necessary for executing the actor: location from where to download the tool, the name of the tool-info module to assemble the command line and parse tool output, additional command-line parameters for the tool, resource limitations to enforce, etc. Listing 2 shows the actor definition file for UAutomizer [32]: the actor name is `uautomizer`, the identifier for the BenchExec tool-info module is

```yaml
actor_name: uautomizer
toolinfo_module: "ultimateautomizer.py"
archive:
  doi: "10.5281/zenodo.3813788"
  spdx_license_identifier: "LGPL-3.0-or-later"
options: ['--full-output', '--architecture', '32bit']
resourcelimits:
  memlimit: "15 GB"
  timelimit: "15 min"
  cpuCores: "8"
format_version: '1.1'
```

Listing 2: Definition of atomic actor in YAML format

`ultimateautomizer`, the DOI of the tool archive (or the URL for obtaining the tool archive), the SPDX license identifier, the options passed by CoVeriTeam to UAutomizer, and resource limits for the execution of the actor. Once an atomic actor has been constructed using an actor definition, CoVeriTeam has all the information necessary to execute the underlying tool with the provided artifacts.

Composition of an Actor. The second key part is the composition of an actor. Lines 6 and 8 in Listing 1 create composite actors using ITE and SEQUENCE, respectively. It is these compositions that create the validating verifier of our running example. Verification actors in CoVeriTeam can exchange information (artifacts) with other actors and cooperate through compositions.

Creation of an Artifact. The notion of artifact in CoVeriTeam is a *file* wrapped in an *artifact type*. The underlying files are the basis of an artifact—exchangeable information. Lines 11 and 12 in Listing 1 create artifacts using the *ArtifactFactory* by providing the *ArtifactType* and the artifact file. These artifacts would then be provided to the executor that then executes the actors on them.

Code Generation. The code generator of CoVeriTeam translates the input program to Python code that uses the internal API of CoVeriTeam. It is a canonical transformation in which the statements for creation of actors and artifacts are converted to Python statements instantiating corresponding classes from the API. Similarly, statements for composition and execution of actors are also transformed.

Execution. Analogously to the construction of actors, the execution of an actor in CoVeriTeam is also divided in two: atomic and composition. Line 15 in Listing 1 executes the actor `validating_verifier` on the given input artifacts.

Figure 6 shows the actor executor for both atomic and composite actors. It executes an actor on the provided artifacts. At first it type checks the inputs, i.e., check if the input types provided to actor comply with the expected input types of the actor. It then calls the executor for atomic or composite actor depending on the actor type. Thereafter, it type checks the outputs, and at last returns the artifacts.

Execution of an atomic actor means the execution of the underlying tool on the provided artifacts. At first, the executor downloads the tool if necessary. CoVeriTeam downloads and unzips the archive that contains the tool on the first execution of an atomic actor. It keeps the tool available in cache for later

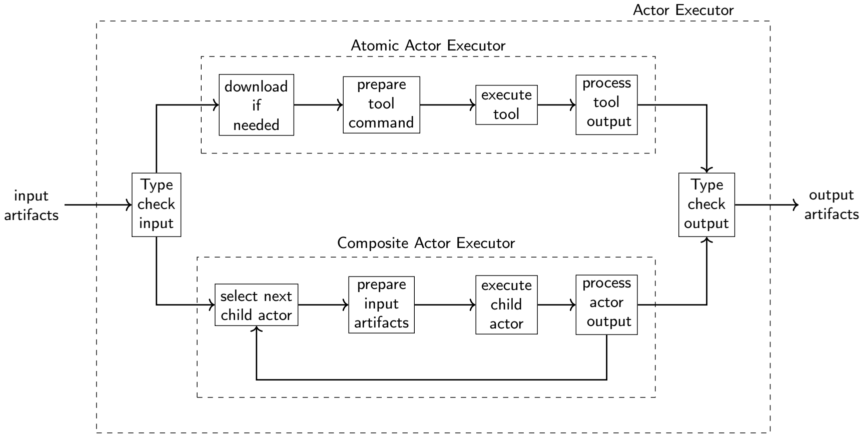


Fig. 6: Abstract view of an actor execution in CoVeriTeam

executions. After this step, the command line for the tool is prepared using the tool-info module. It then executes the tool in a container, and then processes the tool output, i.e., extracts the artifacts from the tool output and saves them.

Execution of a composition means executing the composed actors—making information produced by one available to other during the execution—as per the rules of composition. The composite-actor executor at first selects the next child actor to execute. It then computes the inputs for this selected actor. Then it executes this actor, which can be atomic or another composite actor, on these inputs. It then processes the outputs produced by the execution of the selected child actor. This processing could be temporarily saving, filtering, or modifying the produced artifacts. If needed, it then proceeds to execute the next child actor, otherwise exits the composition execution.

Output. CoVeriTeam collects all the artifacts produced during the execution of an actor, and saves them. The output can be divided into three parts: execution trace, artifacts, and log files. An execution trace is an XML file containing information about the artifacts consumed and produced by each actor, and also the resources consumed by atomic actors (as measured by BenchExec) during the execution. CoVeriTeam also saves the artifacts produced during the execution of an actor. Additionally, for each atomic actor execution, it also saves a log file containing the command which was actually executed and the messages printed on *stdout*.

3.4 API

In addition to the above described features, CoVeriTeam exposes an API that is extensible. We expose actors, artifacts, utility actors, and compositions through Python packages. In this section, we briefly discuss this API.

Library of Actors and Compositions. CoVeriTeam provides a library of some actors and a few compositions that can be instantiated with suitable actors. We considered actors based on the tools participating in the competitions on software verification and testing [5, 6] (available in the replication archives), because those are known to be mature and stable.

The library of compositions contains a validating verifier, an execution-based validator [11], a reducer-based construction of a conditional model checker [15], CondTest [18], and MetaVal [21]. These are present in the `examples/` directory of the CoVeriTeam repository. We discuss some of these constructions in Sect. 4.1.

New Actors, Artifacts, and Tools. New actors, artifacts, and tools can be integrated easily in CoVeriTeam. The integration of a new atomic actor requires only creating a YAML actor definition and, if not already available, implementing a tool-info module. The integration of a new actor type in the language requires (1) creating a class for the actor specifying its input and output artifact types, (2) preparing the parameters to be passed to tool-info module, that in turn would create a command line for the tool execution, using the options from the YAML actor definition, and (3) creating output artifacts from the output files produced by the execution of an atomic actor of that type.

Integration of a new artifact requires creating a new class for the artifact. A basic artifact requires a path containing the artifact. Some artifacts support special features, for example, a test suite is a mergeable artifact (i.e., two test suites for a given input program can be merged into one test suite).

Integrating a new tool in the framework requires: (1) creating the tool-info module for it, (2) creating an actor definition for the tool, (3) providing a self-contained archive that can be executed on a Ubuntu machine.

At present, CoVeriTeam supports all verifiers and validators that are listed on the 2021 competition web sites of SV-COMP[2] and Test-Comp[3]. One needs only a few hours to create a new tool-info module and an actor-definition file. Within a couple of hours we were able to create the actor definitions for about 40 tools participating in SV-COMP and Test-Comp.

4 Evaluation

We now present our evaluation of CoVeriTeam. It consists of a few case studies, and insights from the experiments to measure performance overhead.

4.1 Case Studies

We evaluated CoVeriTeam on four more case studies, as indicated in the fourth column of Table 1. We now explain two of these case studies using figures for compositions. The programs and explanations for all of the case studies are also available in our project repository (linked from the last column of Table 1).

Conditional Testing à la CondTest. Conditional testing [18] allows cooperation between different test generators (testers) by sharing the details of the

[2] https://sv-comp.sosy-lab.org/2021/systems.php

[3] https://test-comp.sosy-lab.org/2021/systems.php

Table 1: Examples of cooperative techniques in the literature

Technique	Year	Reference	Case Study	More Info
Counterexample Checking [38]	2012	Sect. 5		
Conditional Model Checking [13]	2012	Sect. 5		
Precision Reuse [19]	2013	Sect. 5		
Witness Validation [8, 10]	2015, 2016	Figure 4	✓	Sect. 3.3
Execution-Based Validation [11]	2018	Sect. 5	✓	More info
Reducer [15]	2018	Sect. 5	✓	More info
CoVeriTest [14]	2019	Sect. 5		
CondTest [18]	2019	Figures 7 and 8	✓	More info
MetaVal [21]	2020	Figure 9	✓	More info

already covered test goals. A conditional tester outputs a condition, in addition to the generated test suite, representing the work already done. Then this condition is passed as an input to another conditional tester, in addition to the program and test specification. This tester can then focus on only the uncovered goals.

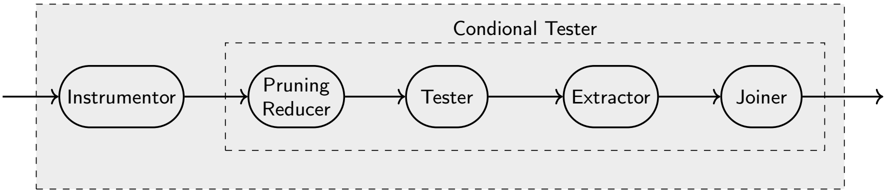


Fig. 7: Design of a conditional tester in CoVeriTeam

Conditional testers can be constructed from off-the-shelf testers [18] with the help of three tools: a reducer, an extractor, and a joiner. A reducer used in conditional testing (Program × Specification × Condition → Program) produces a residual program with the same behavior as the input program with respect to the remaining test goals. A set of test goals represents the condition. An extractor (Program × Specification × TestSuite → Condition) extracts the condition —a set of test goals— covered by the provided test suite.

Figure 7 shows the composition of a conditional tester. First, the reducer produces the reduced program. The composition here uses a *pruning* reducer, which prunes the program according to the covered goals. Second, the tester generates the test cases. Third, the extractor extracts the goals covered in these test cases. Forth, the joiner merges the previously and newly covered goals. The reducer that we used expects the input program to be in a format containing certain labels for the purpose of tracking test goals. So, we put an instrumentor that *instruments* the test specification into the program, by adding these labels.

The conditional-testing concept can also be used iteratively to generate a test suite using a tester based on a verifier [18]. Such a composition uses a verifier as a backend and transforms a counterexample generated by the verifier to a test case. Figure 8 shows the construction of a cyclic conditional tester. In this case, the *tester* itself is a composition of a verifier and a tool, *Witness2Test*, which generates test cases based on a witness produced by a verifier. This tester, in composition

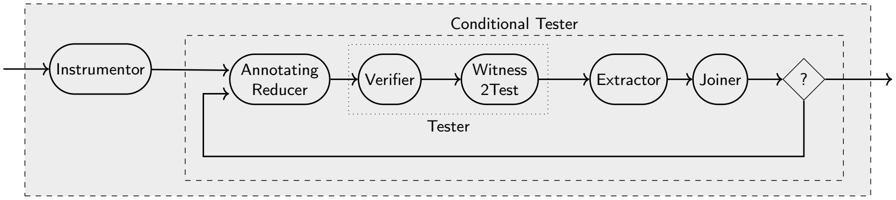


Fig. 8: Design of a cyclic conditional tester in CoVeriTeam

with a reducer, extractor, and a joiner is our conditional tester. This construction uses an *annotating* reducer, which (i) annotates the program with *error* labels for the verifier to find the path to and (ii) filters out the already covered goals, i.e., the condition, from the list of goals to be annotated. We put the conditional tester in the REPEAT composition to execute iteratively. The composition tracks the set *'covered_goals'* to detect the fixed point to decide termination of the iteration. This composition will keep on accumulating the test suite generated in each iteration and finally output the union of all the generated test suites (see Sect. 3.1). As above, an instrumentor is placed before the conditional tester.

Verification-Based Validation à la MetaVal. MetaVal [21] uses off-the-shelf verifiers to perform validation tasks. A validator (Program × Specification × Verdict × Witness → Verdict × Witness) validates the result produced by a verifier. MetaVal employs a three-stage process for validation. In the first stage, MetaVal instruments the input program with the input witness. The instrumented program —a product of the witness and the original program— is equivalent to the original program modulo the provided witness. This means that the instrumented program can be given to an off-the-shelf verifier for verification; and this verification functions as validation. In the second stage, MetaVal selects the verifier to use based on the specification. It chooses CPAchecker for reachability, UAutomizer for integer overflow and termination, and Symbiotic for memory safety.[4] In the third stage, the instrumented program is fed to a verifier along with the specification for verification. If the verification produces the expected result, then the result is confirmed and the witness valid, otherwise not.

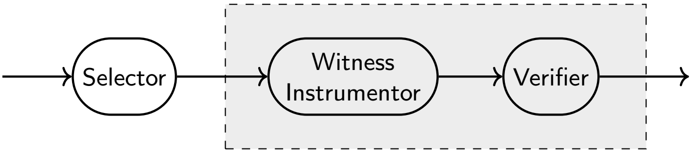


Fig. 9: Design of MetaVal in CoVeriTeam

Figure 9 shows the construction of MetaVal. First, the selector is executed that selects the backend verifier to execute. After this step, the program is

[4] These were the best performing tools for a property according to SV-COMP results.

instrumented with the witness, and then the instrumented program is given to the selected verifier for checking the specification.

4.2 Performance

CoVeriTeam is a lightweight tool. Its container mode causes an overhead of around 0.8 s for each actor execution in the composition, and the tool needs about 44 MB memory. This means that if we run a tool 10 times in a sequence in a shell script unprotected and compare this to using the sequence composition in CoVeriTeam in protected container mode on the same input, the execution using CoVeriTeam will take 8 s longer and requires 44 MB more memory. In our experience, this overhead is not an issue for verification as, in general, the time taken for verification dominates the total execution time. For short-running, high-performance needs, the container mode can be switched off. We have conducted extensive experiments for performance evaluation of CoVeriTeam and point the reader to the supplementary webpage for this article for more details.

5 Related Work

We divide our literature overview into two parts: approaches for tool combinations, and cooperative verification approaches.

Approaches for Tool Combinations. *Evidential Tool Bus* (ETB) [29, 30, 39] is a distributed framework for integration of tools based on a variant of Datalog [1, 24]. It stores the established claims along with the corresponding files and their versions. This allows the reuse of partial results in regression verification. ETB orchestrates tool interaction through scripts, queries, and claims.

Our work seems close to ETB on a quick glance, but on a closer look there are profound differences. Conceptually, ETB is a query engine that uses claims, facts, and rules to define and execute a workflow. Whereas, CoVeriTeam has been designed to create and execute actors based on tools and their compositions. We give some semantic meaning, arguably simplistic, to the tools using (i) wrapper types of artifacts for the files produced and consumed by a tool and (ii) the notion of *verification actors* that allows us to see a tool as a *function*. This allows us to type-check tool compositions and allow only well-defined compositions. On the implementation side, we support more tools. This task was simplified by our design choice to use the integration mechanisms provided by BenchExec (as used in SV-COMP and Test-Comp). Most well known automated verification tools already have been integrated in CoVeriTeam.

Electronic Tools Integration platform (ETI) [40] was envisioned as a "one stop shop" for the experimentation and evaluation of tools from the formal-methods community. It was intended to serve as a tool presentation, tool evaluation, and benchmarking site. The idea was to allow users to access tools through the internet without the need to install them. An ETI user is expected to provide an LTL based specification, based on which an execution scheme is synthesized.

The key focus of ETI and its incarnations has been remote tool execution, and their integration over internet. The tools are viewed agnostic to their function. We, in contrast, (i) have tackled local execution concerns and (ii) see a tool in its function as an actor that consumes and produces certain kinds of artifacts. The semantic meaning of a tool is given by this role.

Cooperative Verification Approaches. Our work targets developing a framework to express and execute cooperative verification approaches. In this section we describe some of these approaches from literature. We have implemented some of these combinations in CoVeriTeam, some of which are described in Sect. 4.

A reduction of the input program using the counterexample produced by a verifier was discussed [38], where the key idea is to use the counterxample to provide the variable assignments to the program.

Conditional model checking (CMC) [13] outputs a condition —a summary of the knowledge gained— if the model checker fails to produce a verdict. The condition allows another model checker to save the effort of looking into already explored state space. Reducers [15] can turn any off-the-shelf model checker into a conditional model checker. Reducers take a source program and a condition and produce a *residual program* whose paths cover the unverified state space (negation of the condition). Conditional testing [18] applies the principle of conditional model checking to testing. A conditional tester outputs, in addition to the generated test cases, the goals for which test cases have been generated.

The idea of reusing the knowledge about already done work to reduce the workload of another tool was also applied to combine program analysis and testing [25, 31, 35]. One of these approaches [31] is based on conditional model checking [13]. In this case, the condition is used to construct a residual program, which is then fed to a test-case generator. Another approach [25] instruments the program with assumptions and assertions describing the already completed verification work. Then a testing tool is used to test the assumptions. Program partitioning [35] first performs the testing and then removes the satisfactorily tested paths and verifies the rest. CoVeriTest [14], cooperative verifier-based testing, is a tester based on cooperation between different verification-based test-generation techniques. CoVeriTest uses conditional model checkers [13] as verifier backends.

Precision reuse [19] is based on the use of abstraction precisions. The precision of an abstract domain is a good candidate for cooperation because it is small in size, and represents important information, i.e., the level of abstraction at which the analysis works. A model checker in addition to producing a verdict also produces a file containing information specifying precision, e.g., predicates.

Model checkers can also produce a witness, in addition to the verdict, as a justification of the verdict. These witnesses could be counterexamples for violations of a safety property, invariants as a proof of a safety property, a lasso for non-termination, a ranking function for termination, etc. These witnesses can be used later to help validate the result produced by a verifier [8, 9, 10].

Execution-based result validation [11] uses violation witnesses to generate test cases. A violation witness of a safety specification is refined to a test case. The test case is then used to validate the result of the verification.

6 Conclusion

Due to the free availability of many excellent verifiers, the time is ripe to view verification tools as components. It is necessary to have standardized interfaces, in order to define the inputs and outputs of verification components. We have identified a set of verification artifacts and verification actors, and a programming language for on-demand construction of new, combined verification systems.

So far, the architectural hierarchy ends mostly at the verifiers: verifiers are based on SMT solvers, which are based on SAT solvers, which are based on data-structure libraries. CoVeriTeam wants to change this and use verification artifacts as first-class objects in specifying new verifiers. We show on a few selected examples how easy it is to construct some verification systems that were so far hard-coded using glue code and wrapper scripts. We hope that many researchers and practitioners in the verification community find it interesting and stimulating to experiment on a high level with verification technology.

Future Work. The approach of CoVeriTeam opens up a whole new area of possibilities that yet needs to be explored. We have identified three key areas for the further work: (i) remote execution of tools, (ii) policy specification and enforcement, and (iii) more compositions and combinations. CoVeriTeam provides an interface for a verification tool based on its behavior. A web service wrapped around CoVeriTeam can be used to delegate execution of an actor, hence verification work, to the host of the service. The client for such a service can be transparently integrated in CoVeriTeam. In fact, we already provide client integration for a restricted and experimental version of such a service. Also, a user executing a combination of tools might want to have some restrictions on which tools should be allowed to execute. For example, a user might want to execute only those tools that comply with a certain license, or only those tools that are downloaded from a trusted source. A cooperative verification tool should support the specification and enforcement of such user *policies*. Further, we plan to support more compositions for cooperative verification in CoVeriTeam as we come across them. Recently, we were working on a *parallel-portfolio* composition [17].

Declarations

Data Availability Statement. CoVeriTeam is publicly available under the Apache 2 license.[5] The data from our performance evaluation is available at the supplementary webpage of the paper.[6] A replication package including all evaluated external tools is available at Zenodo [16].

Funding Statement. This work was funded in part by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop).

Acknowledgement. We thank Thomas Lemberger and Philipp Wendler for their valuable feedback on this article, and the SV-COMP community for their valuable feedback on experimenting with CoVeriTeam.

[5] https://gitlab.com/sosy-lab/software/coveriteam/

[6] https://www.sosy-lab.org/research/coveriteam/

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with Slam. Commun. ACM **54**(7), 68–76 (2011). https://doi.org/10.1145/1965724.1965743
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., University of Iowa (2015), available at www.smt-lib.org
5. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
6. Beyer, D.: Second competition on software testing: Test-Comp 2020. In: Proc. FASE. pp. 505–519. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_25
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. LNCS 13244, Springer (2022)
8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022)
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
11. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
12. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. ACM (2022)
13. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
14. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
15. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
16. Beyer, D., Kanav, S.: Reproduction package for article 'CoVeriTeam: On-demand composition of cooperative verification systems'. Zenodo (2021). https://doi.org/10.5281/zenodo.5644953
17. Beyer, D., Kanav, S., Richter, C.: Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In: Proc. FASE. Springer (2022)
18. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11

19. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). https://doi.org/10.1145/2491411.2491429
20. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
21. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
22. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
23. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
24. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. Knowledge and Data Eng. **1**(1), 146–166 (1989). https://doi.org/10.1109/69.43410
25. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13
26. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). https://doi.org/10.1007/10722167_15
27. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
28. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL. pp. 269–282. ACM (1979). https://doi.org/10.1145/567752.567778
29. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI. pp. 275–294. LNCS 7737, Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_18
30. Cruanes, S., Heymans, S., Mason, I., Owre, S., Shankar, N.: The semantics of Datalog for the Evidential Tool Bus. In: Specification, Algebra, and Software. pp. 256–275. Springer (2014). https://doi.org/10.1007/978-3-642-54624-2_13
31. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
32. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
33. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(7), 51–54 (1997). https://doi.org/10.1126/science.275.5296.51
34. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154**(16), 2291–2306 (2006). https://doi.org/10.1016/j.dam.2006.04.015
35. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: Proc. WODA. pp. 11–16. ACM (2006). https://doi.org/10.1145/1138912.1138916

36. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
37. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
38. Rocha, H.O., Barreto, R.S., Cordeiro, L.C., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Proc. IFM. pp. 128–142. LNCS 7321, Springer (2012). https://doi.org/10.1007/978-3-642-30729-4_10
39. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM. pp. 36–36. LNCS 3785, Springer (2005). https://doi.org/10.1007/11576280_3
40. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1-2), 9–30 (1997). https://doi.org/10.1007/s100090050003

Author Index