CONFERENCE SERIES: FORMAL METHODS IN COMPUTER-AIDED DESIGN 5

Nina Narodytska / Philipp Rümmer (Eds.)

PROCEEDINGS OF THE 24TH CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2024

TU WIEN Academic Press

fmcad.24

Nina Narodytska / Philipp Rümmer (Eds.)
PROCEEDINGS OF THE 24TH CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2024

Conference Series: Formal Methods in Computer-Aided Design
Volume 5

Conference Series: Formal Methods in Computer-Aided Design

Series edited by:
Warren A. Hunt, Jr., The University of Texas at Austin
Austin, TX 78705 | hunt@cs.utexas.edu
Georg Weissenbacher, TU Wien
Karlsplatz 13, 1040 Vienna, Austria | georg.weissenbacher@tuwien.ac.at

The Conference on Formal Methods in Computer-Aided Design (FMCAD) is an annual conference on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.

Information on this publication series and the volumes published therein is available at www.tuwien.ac.at/academicpress.

Volume 4 edited by:
Nina Narodytska, VMware by Broadcom, Palo Alto, USA | n.narodytska@gmail.com
Philipp Rümmer, University of Regensburg, Germany and Uppsala University, Sweden | philipp.ruemmer@ur.de

Nina Narodytska / Philipp Rümmer (Eds.)

PROCEEDINGS OF THE 24TH CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2024

TU WIEN Academic Press

Cite as:
Narodytska, N., & Rümmer, P. (Eds.). (2024). *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024*. TU Wien Academic Press. https://doi.org/10.34727/2024/isbn.978-3-85448-065-5

TU Wien Academic Press, 2024

c/o TU Wien Bibliothek
TU Wien
Resselgasse 4, 1040 Wien
academicpress@tuwien.ac.at
www.tuwien.at/academicpress

ISBN (online): 978-3-85448-065-5
ISSN (online): 2708-7824

Available online: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5

Media proprietor: TU Wien, Karlsplatz 13, 1040 Wien
Publisher: TU Wien Academic Press
Publication series editor: Warren A. Hunt, Jr. and Georg Weissenbacher
Editors (responsible for the content): Nina Narodytska and Philipp Rümmer

Preface

These are the proceedings of the twenty-fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD), which was held in Prague, Czech Republic, October 14–18, 2024. The first FMCAD was organized in 1996, and FMCAD was a bi-annual conference until 2006, when the FMCAD and CHARME conferences merged into a single FMCAD. Since then, FMCAD has been an annual event. FMCAD 2024 was the twenty-fourth edition in the series, covering formal aspects of computer-aided system design including verification, specification, synthesis, and testing. It provided a leading forum to researchers in academia and industry to present and discuss groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. The program of FMCAD 2024 consisted of one tutorial, three invited talks, the presentation of the Hardware Model Checking Competition HWMCC'24, a student forum, and the main program consisting of presentations of 29 accepted peer-reviewed papers. FMCAD 2024 was co-located with the VSTTE 2024 conference, when took place on October 14–15.

The joint VSTTE/FMCAD tutorial day (October 15) featured two tutorials:

- The VSTTE tutorial: *The Lean Programming Language and Theorem Prover,* given by Sebastian Ullrich and Joachim Breitner;
- The FMCAD tutorial: *Writing proofs in Dafny,* given by Rustan Leino.

The main FMCAD conference (October 16–18) featured three invited talks:

- *Tackling Scalability Issues in Bit-Vector Reasoning* by Aina Niemetz;
- *Some Adventures in Learning Proving, Instantiation and Synthesis* by Josef Urban;
- *Harnessing SMT Solvers for Reasoning about DeFi Protocols* by Mooly Sagiv.

FMCAD 2024 received 56 submissions, out of which the committee decided to accept 29 for publication. Each submission received at least four reviews. The topics of the accepted papers include machine learning, model checking, hardware and software validation, SAT&SMT solving and proofs generation. Among the accepted papers, there are 26 regular papers (23 long and 3 short) and 3 tool/case study papers (all short). FMCAD 2024 hosted the twelfth edition of the FMCAD Student Forum, which has been held annually since 2013 and provides a platform for graduate students at any career stage to introduce their research to the FMCAD community. The FMCAD Student Forum 2024 was organized by Martin Blicha and Nestan Tsiskaridze and featured short presentations of 23 accepted contributions. The proceedings provide a detailed description of the Student Forum and lists all accepted contributions.

FMCAD 2024 was made possible by the support of a large number of people, as well as our sponsors. The program committee members and additional reviewers, listed on the following pages, did an excellent job providing detailed and insightful reviews. The reviews helped us build a strong program and helped the authors improve their submissions. We thank each and everyone of them for dedicating their time and providing their expertise. We would like to thank the local organization chair, Mikoláš Janota, and the registration chair, Milena Zeithamlová, who did an amazing job taking care of the organization and all practical matters. We thank our web master Julie Cailler, our sponsorship chair Guy Amir, and the Student Forum organizers Martin Blicha and Nestan Tsiskaridze. We thank the organizers of the HWMCC competition, Armin Biere, Nils Froleyks, and Mathias Preiner. We thank Georg Weissenbacher, both for his exceptional assistance in organizing the event, communicating to us the decisions of the steering committee, as well as being the publication chair.

Holding a conference like FMCAD would not be feasible without the financial support of our sponsors. We would like to express our gratitude to the sponsors, given here in alphabetical order: AWS, Cadence, General Electric Aerospace, Intel, NSF, Toyota, and VMware by Broadcom.

Last but not least, we thank all authors who submitted their papers to FMCAD 2024, and whose contributions and presentations form the core of the conference. The conference proceedings are available as Open Access Proceedings published by TU Wien Academic Press, and through the IEEE Xplore Digital Library.

We are grateful to everyone who presented their paper, gave a keynote or gave a tutorial. We thank all attendees of FMCAD for supporting the conference and making FMCAD an engaging and enjoyable event.

October 2024

Nina Narodytska VMware by Broadcom, USA
Philipp Rümmer University of Regensburg, Germany and Uppsala University, Sweden

Organizing Committee

Program Co-Chairs

Nina Narodytska	VMware Research by Broadcom, CA, USA
Philipp Rümmer	University of Regensburg, Germany, and Uppsala University, Sweden

Local Organization Chair

Mikoláš Janota	Czech Technical University in Prague, Czech Republic

Registration Chair

Milena Zeithamlová	Action M Agency, Prague, Czech Republic

Student Forum Chairs

Martin Blicha	Università della Svizzera italiana, Switzerland
Nestan Tsiskaridze	Stanford University, CA, USA

Sponsorship Chair

Guy Amir	Cornell University, NY, USA

Web Chair

Julie Cailler	University of Regensburg, Germany

Publication Chair

Georg Weissenbacher	TU Wien, Austria

FMCAD Steering Committee

Board of the FMCAD Association

Program Committee

FMCAD 2024 Program Committee

Nina Narodytska (co-chair)	VMware Research by Broadcom
Philipp Rümmer (co-chair)	University of Regensburg
Guy Amir	Cornell University
Mohamed Faouzi Atig	Uppsala University
Jaroslav Bendík	Certora
Armin Biere	University of Freiburg
Per Bjesse	Synopsys Inc.
Nikolaj Bjørner	Microsoft
Roderick Bloem	Graz University of Technology
Shaowei Cai	Chinese Academy of Sciences
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Rohit Dureja	Advanced Micro Devices, Inc.
Gabriel Ebner	Microsoft Research
Grigory Fedyukovich	Florida State University
Alberto Griggio	Fondazione Bruno Kessler
Arie Gurfinkel	University of Waterloo
Liana Hadarean	Amazon Web Services
William Harrison	Idaho National Laboratory
Bo-Yuan Huang	Intel Corporation
William Hung	Cadence
Warren Hunt	The University of Texas at Austin
Ahmed Irfan	SRI International
Mikoláš Janota	Czech Technical University in Prague
Daniela Kaufmann	TU Wien
Tim King	Google
Anna Lukina	TU Delft
Andreas Lööw	Imperial College London
Ravi Mangal	Colorado State University
Ken McMillan	UT Austin
Baoluo Meng	GE Aerospace Research
David Monniaux	CNRS / VERIMAG
Alexander Nadel	Technion & Intel
Ruzica Piskac	Yale University
Mathias Preiner	Stanford University
Mohammad Rahmani Fadiheh	Stanford University
Andrew Reynolds	University of Iowa
Kristin Yvonne Rozier	Iowa State University
Christoph Scholl	University of Freiburg
Natasha Sharygina	University of Lugano, Switzerland
Aditya A. Shrotri	Siemens Digital Industries Software
Carsten Sinz	Karlsruhe University of Applied Sciences
Christoph Sticksel	The MathWorks

Martin Suda	Czech Technical University in Prague
Tachio Terauchi	Waseda University
Yakir Vizel	The Technion
Tomáš Vojnar	Brno University of Technology
Mike Whalen	AWS
Thomas Wies	New York University
Hongce Zhang	Hong Kong University of Science and Technology (Guangzhou)
Shufang Zhu	University of Liverpool
Florian Zuleger	TU Wien
Ivana Černá	Masaryk University

FMCAD 2024 Student Forum Committee

Martin Blicha (co-chair)	Università della Svizzera italiana
Nestan Tsiskaridze (co-chair)	Stanford University
Guy Amir	Cornell University
Haniel Barbosa	Universidade Federal de Minas Gerais
Armin Biere	University of Freiburg
Nikolaj Bjørner	Microsoft
William Eiers	Stevens Institute of Technology
Katalin Fazekas	TU Wien
Alberto Griggio	Fondazione Bruno Kessler
Arie Gurfinkel	University of Waterloo
Petra Hozzová	Czech Technical University in Prague
Antti Hyvärinen	Certora
Ahmed Irfan	SRI International
Konstantin Korovin	University of Manchester
Daniel Larraz	University of Iowa
Ondřej Lengál	Brno University of Technology
Alexander Nadel	Technion & Intel
Andres Noetzli	Stanford University
Rodrigo Otoni	Università della Svizzera italiana
Sophie Rain	TU Wien
Mark Santolucito	Barnard College, Columbia University
Christoph Sticksel	The MathWorks
Hari Govind V. K.	University of Waterloo & Microsoft
Yoni Zohar	Bar Ilan University

Additional Reviewers

Barbosa, Haniel
Bogaerts, Bart
Britikov, Konstantin
Brown, Chad

Cailler, Julie
Chadha, Rohit
Chvalovský, Karel

Dewes, Rafael

Esen, Zafer

Fazekas, Katalin
Feng, Jincao
Fleury, Mathias

Gauthier, Thibault
Govind, R

Hamza, Ameer
He, Fei
Herrmann, Roland
Hinnerichs, Tilman
Holík, Lukáš
Hu, Guangyu

Isac, Omri

Kern, Philipp
Kolárik, Tomáš
Konrad, Alexander

Labbaf, Faezeh
Lengal, Ondrej
Li, Elaine
Lipparini, Enrico
Lutz, Sterre

Maderbacher, Benedikt
Mony, Hari

Paul, Saswata

Rao, Vikas
Rebola Pardo, Adrian
Riley, Daniel
Rodriguez, Andoni
Rogalewicz, Adam

Saivasan, Prakash
Seufert, Tobias
Sextl, Florian
Sindoni, Giulia

Temel, Mertcan

Varanasi, Sarat Chandra

Zavalia, Lucas

Table of Contents

Formal Methods in Computer-Aided Design 2024

Writing Proofs in Dafny

K. Rustan M. Leino
Amazon Web Services
Seattle, WA
leino@amazon.com

***Abstract*—Dafny is a verification-aware programming language. In a nutshell, the language is Java-like and has support for writing specifications and proofs. It has a long history of being used in education, has been the cornerstone of several ambitious research projects, and is in industrial use at AWS.**
This tutorial teaches how to write various kinds of proofs in Dafny. It covers proofs of imperative and functional programs, as well as the formalization of models and mathematical proofs. It demonstrates different proof styles and shows how to think about and debug proofs in the Dafny setting.
The tutorial does not assume any prior experience in using Dafny or other proof assistants.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_1

Tackling Scalability Issues in Bit-Vector Reasoning

Aina Niemetz
Stanford University
Stanford, CA
niemetz@cs.stanford.edu

***Abstract*—Efficiently reasoning about bit-vector constraints in Satisfiability Modulo Theories (SMT) has been an ongoing challenge for many years. The dominant state-of-the-art approach for solving bit-vector formulas in SMT is bit-blasting, an eager reduction to propositional logic that is typically combined with aggressive simplifications of the input constraints prior to the actual reduction step. Even though this eager reduction may come at the cost of significantly increasing the formula size, it is surprisingly efficient in practice—thanks to state-of-the-art SAT solvers, which are usually able to efficiently deal with complex formulas over millions of variables. This size increase, however, is a potential bottleneck and the main reason why bit-blasting does not generally scale well for increasing bit-widths, especially in the presence of arithmetic operators, which translate to large and complex Boolean circuits on the bit-level.**

To tackle these scalability issues, there are two (orthogonal) avenues to explore: developing alternative approaches that do not (mainly) rely on translations to the SAT level, and improving the scalability of bit-blasting itself. In this talk, we will highlight techniques in each category: a propagation-based local search procedure as an alternative to bit-blasting, which can only determine satisfiability but improves performances over bit-blasting on satisfiable instances, and a CEGAR-style abstraction-refinement procedure that significantly improves the scalability of bit-blasting. We extended the state-of-the-art SMT solver Bitwuzla with both techniques and show that they significantly improve solver performance on a variety of benchmark sets across all logics supported by Bitwuzla, including combinations of bit-vectors with arrays, uninterpreted functions and floating-point arithmetic.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_2

Some Adventures in Learning Proving, Instantiation and Synthesis

Josef Urban
Czech Institute of of Informatics, Robotics and Cybernetics
Prague, Czech Republic
josef.urban@cvut.cz

***Abstract*—Human problem solvers often combine deductive reasoning and exploration with learning and pattern matching. In the recent years such combinations are also increasingly developed for building stronger automated theorem provers, SMT solvers and conjecturing and synthesis systems.**
The methods in this field include equipping the current deductive systems with efficient statistically learned guidance that controls the choice of the inference steps, using for example fast decision trees, graph neural networks and their combinations.
Learning and AI methods can also be used to automatically design new symbolic strategies for today's ATPs and SMTs. This has the advantage of producing explainable ideas for steering the search space, which can be further taken up and modified by the systems' developers.
I will also discuss several methods that try to directly synthesize reasoning objects such as instantiations and OEIS explanations, using various neural approaches.
Perhaps the most interesting aspect of this research are the positive feedback loops between the proving and the learning methods. I will show that some of them can today go quite far and create quite interesting "alien" solutions.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_3

Harnessing SMT Solvers for Reasoning about DeFi Protocols

Mooly Sagiv
Certora and Tel Aviv University
Tel Aviv, Israel
msagiv@acm.org

***Abstract*—DeFi (Decentralized Financial) Protocols implement financial programs using low-level programming. DeFi adoption started to go parabolic in 2020, and it's still very robust in different market conditions in 2024. Today, DeFi assets exceed 300 billion USD. A fundamental principle behind DeFi is that small open-source software called "smart contracts" precisely define the trading conditions and create an open global economy not controlled by governments and people.**

However, smart contracts are difficult to implement correctly since their behavior can radically change in different market conditions. Moreover, hackers constantly try to abuse the code to drain the money stored in the smart contracts. On the positive side, it is pretty natural to write high-level formal specifications of smart contracts since their economical utilities are well understood. Indeed, this is a unique domain where software developers are eager to write formal specifications.

I will describe the challenges of harnessing existing SMT solvers for reasoning about smart contracts.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_4

The FMCAD 2024 Student Forum

Martin Blicha
University of Lugano & Ethereum Foundation
Prague, Czechia
martin.blicha@usi.ch

Nestan Tsiskaridze
Stanford University
Stanford, USA
nestan@stanford.edu

***Abstract*—The Student Forum at the International Conference on Formal Methods in Computer–Aided Design (FMCAD) gives undergraduate and graduate students the opportunity to introduce their research to the Formal Methods community and receive feedback. In 2024, the event took place in Prague, Czechia. Twenty three students were invited to give a short talk and present a poster of their work.**

Since 2013, the FMCAD Student Forum provides a platform for undergraduate and graduate students at any career stage to present their research to the audience of the FMCAD conference. The 2024 edition of the FMCAD Student Forum follows the tradition of its predecessors, which took place in:

- Portland, Oregon, USA in 2013 [1]
- Lausanne, Switzerland in 2014 [2]
- Austin, Texas in 2015 [3] and 2018 [4]
- Mountain View, California, USA in 2016 [5]
- Vienna, Austria in 2017 [6]
- San Jose, California, USA in 2019 [7]
- Virtual in 2020 [8] and 2021 [9]
- Trento, Italy in 2022 [10]
- Ames, Iowa, USA in 2023 [11]

FMCAD 2024 hosted the twelves edition of the Student Forum. Graduate and undergraduate students were invited to submit two-page reports of their current research and ongoing work in the scope of the FMCAD conference. There were 24 submissions to the forum, 23 of them were accepted one of which was withdrown. The Student Forum program committee reviews were based on the overall quality, novelty of the work, its potential impact on the Formal Methods community, as well as the potential positive impact on the student to have the opportunity to participate in the forum. The accepted submissions covered a wide range of topics relevant to the FMCAD community, from foundational aspects of automated reasoning, to analysis and verification of software, hardware, and neural networks, as well as applications of formal methods to security and dynamical system. Each submission received 3 reviews. The following contributions have been accepted[1] (excluding the withdrawn contribution):

- Csanád Telbisz and Dániel Szekeres *Correctness Witnesses for Concurrent Software Verification*
- Levente Bajczi and Marian Lingsch-Rosenfeld *Software Verification Witnesses for Weak Memory*
- Levente Bajczi *CHCs for Weak Memory*
- Islam Hamada *Incremental Construction of Inductive Invariants for Model Checking*
- Zsófia Ádám, Levente Bajczi, Marek Jankola and Marian Lingsch-Rosenfeld *Towards Validation of More Expressive Software Non-Termination Witnesses*
- Luke Miga *Verifying Axiomatic Microarchitectural Models in the Coq Proof Assistant*
- Konstantin Britikov *Analysis of Multiloop Programs With Nested Loops Using Transition Power Abstraction*
- Rachel Cleaveland *Theory of Strings in Symbolic Execution*
- Siddharth Priya *Optimizing Rust Programs Using Ownership*
- Daneshvar Amrollahi *Towards Improved Stability for SMT Solvers*
- John Kolesar *Coinductive Proofs of Regular Expression Equivalence in Zero Knowledge*
- Feitong Qiao *Timed Data Types for Hardware*
- Elizaveta Pertseva and Alex Ozdemir *Multimodular Reasoning for Satisfiability Modulo Theories*
- Fuqi Jia *A Theory-Agnostic SMT Sampling Framework*
- Milan Ganai *Hamilton-Jacobi Reachability Estimation*
- Samantha Archer *SymLeak: Quantifying Side Channel Leakage with Symbolic Execution*
- Daniel Mendoza *Towards LLM-assisted hardware verification*
- Edward Wang *Work-in-Progress: An SMT-Based, Correct-by-Construction Place-and-Route Framework*
- Michal Hečko *Automata-based Decision Procedure for Presburger Arithmetic Augmented with Algebraic Reasoning*
- Áron Ricardo Perez-Lopez and Samantha Archer *Word-Level Model Checking with IC3 in Pono*
- Roxana-Mihaela Timon *Verification of a dynamic programming-based algorithm for the Activity Selection Problem in Dafny*
- Márk Somorjai and Mihály Dobos-Kovács *Stack Abstraction for Interprocedural Software Verification*

We formed a program committee to cover a wide range of topics so students could receive expert feedback on their work. The 2024 FMCAD Student Forum program committee consisted of Martin Blicha (co-chair), Nestan Tsiskaridze (co-chair), Guy Amir, Haniel Barbosa, Armin Biere, Nikolaj Bjørner, William Eiers, Katalin Fazekas, Alberto Griggio,

[1]Only student authors listed for brevity.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_5

Arie Gurfinkel, Petra Hozzová, Antti Hyvärinen, Ahmed Irfan, Konstantin Korovin, Daniel Larraz, Ondřej Lengál, Alexander Nadel, Andres Noetzli, Rodrigo Otoni, Sophie Rain, Mark Santolucito, Christoph Sticksel, Hari Govind V. K., and Yoni Zohar.

We would like to thank the organizers of FMCAD, as well as the FMCAD Student Forum program committee, who have made the FMCAD Student Forum possible. We would like to thank FMCAD, NSF, Amazon Web Services, Cadence, GE Aerospace, Intel, Toyota, and VMWare for providing student travel support and making it possible to award travel grants to all students. Additionally, we are grateful to the student authors and their research mentors who have contributed their excellent work to the program.

References

[1] T. Wahl, "The FMCAD graduate student forum," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 16–17. [Online]. Available: https://doi.org/10.1109/FMCAD.2013.7035523

[2] R. Piskac, "The FMCAD 2014 graduate student forum," in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, p. 13. [Online]. Available: https://doi.org/10.1109/FMCAD.2014.6987589

[3] G. Weissenbacher, "The FMCAD 2015 graduate student forum," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, p. 8. [Online]. Available: https://doi.org/10.1109/FMCAD.2015.7542246

[4] D. Jovanović and A. Reynolds, "The FMCAD 2018 graduate student forum," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–1, https://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD18/student-forum/.

[5] H. Hojjat, "The FMCAD 2016 graduate student forum," in *Formal Methods in Computer-Aided Design (FMCAD), 2016*. IEEE, 2016, pp. 8–8, https://fmcad.forsyte.at/FMCAD16/student-forum.html.

[6] K. Heljanko, "The FMCAD 2017 graduate student forum," in *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 2017, pp. 10–10, https://fmcad.org/FMCAD17/student-forum/.

[7] G. Fedyukovich, "The FMCAD 2019 student forum," in *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019, pp. 1–1, https://fmcad.forsyte.at/FMCAD19/student-forum/.

[8] P. Schrammel, "The FMCAD 2020 student forum," in *2020 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2020, pp. 1–1, https://fmcad.forsyte.at/FMCAD20/student-forum/.

[9] M. Santolucito, "The FMCAD 2021 student forum," in *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021, pp. 1–1, https://fmcad.org/FMCAD21/student_forum/.

[10] M. Preiner, "The FMCAD 2023 student forum," in *2022 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2022, pp. 1–1, https://fmcad.org/FMCAD22/student_forum/.

[11] M. Janota and N. Narodytska, "The FMCAD 2023 student forum," in *2023 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2023, pp. 1–1, https://fmcad.org/FMCAD23/student_forum/.

Hardware Model Checking Competition 2024

Armin Biere
University of Freiburg
Freiburg, Germany
biere@cs.uni-freiburg.de

Nils Froleyks
Johannes Kepler University
Linz, Austria
nils.froleyks@jku.at

Mathias Preiner
Stanford University
Stanford, United States
preiner@cs.stanford.edu

***Abstract*—The Hardware Model Checking Competition 2024 (HWMCC'24) was the 12th competitive event for hardware model checking tools. The competition was affiliated to the 24th conference on Formal Methods in Computer-Aided Design 2024 (FMCAD'24), which took place in Prague, Czech Republic, from October 14 to 18, 2024.**

***Index Terms*—Automated Reasoning, Model Checking, Hardware Verification, Word-level Reasoning, Bit-Vectors, Certificates**

The Hardware Model Checking Competition (HWMCC'24) in 2024 is the 12th incarnation in this series of competitive events to evaluate hardware model checking. Since it started in 2007 it was repeated annually with some exceptions. After the previous competition in 2020 the organizers took a break to resume the competition in 2024. The competition in 2024 is affiliated, as most of the time, with the conference on Formal Methods in Computer-Aided Design (FMCAD), which is considered the primary venue for formal hardware verification. Alternatively in 2007, 2008, 2010 and 2014 it was affiliated with the conference of Computer-Aided Verification (CAV).

The previous competition in 2020 continued with word-level tracks, which were introduced in 2019. These word-level tracks focus on bit-vector models with and without arrays in the BTOR2 format [1]. This suggests that model checkers participating in this track should make use of SMT solvers over the theory of bit-vectors. Before 2019 all competition tracks used bit-level models in the AIGER format [2], but were split into safety, multi-property, liveness and deep tracks. Since 2014 and particularly in 2017, the last competition before 2019, the ABC tool [3] dominated almost all bit-level tracks.

One motivation for moving to word-level tracks is the conjecture that SMT solving is more effective than plain SAT solving if the models are given in terms of bit-vectors. However, in 2019 the word-level model checkers could not fulfill this promise and were trailing ABC by a large margin in terms of performance. This was particularly the case for unsatisfiable properties, where a bad state violating the single safety property can not be reached. Note, that ABC was run on AIGER models obtained from the BTOR2 models through bit-blasting, except for the array track, as bit-blasting of BTOR2 models with arrays is difficult. Having arrays, modelling memory or caches, is considered a feature of SMT solvers and should give them an advantage over bit-level reasoning.

In 2020 the picture changed and word-level model checkers started to become competitive to ABC on the bit-vector track without arrays, while not losing their advantage on bit-vector models with arrays, as bit-blasting arrays was still not available. Therefore, the organizers of HWMCC'24 decided to continue both word-level tracks, i.e., with and without arrays.

While the previous competition in 2020 focused on word-level exclusively, the single safety property track came back in 2024. However, as a novel feature, participating model checkers are required to produce model-checking certificates. These certificates were actually AIGER circuits and should have an inductive property. They further need to simulate the original circuit as formalized in [4], [5], [6]. The tool CERTIFAIGER is used to check both requirements using SAT solvers. The goal of the certified track is to increase trust in verification results produced by model checkers, following the success story of proof producing SAT solvers in both academia and industry, e.g., producing proofs became mandatory in the main track of the SAT competition in 2016 [7].

More details on the competition, including provided tools, submission procedure and deadlines, the results and their presentation are available at the competition home page [8].

REFERENCES

[1] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , BtorMC and Boolector 3.0," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 587–595.

[2] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.

[3] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. B. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40.

[4] E. Yu, N. Froleyks, A. Biere, and K. Heljanko, "Stratified certification for k-induction," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 59–64.

[5] ——, "Towards compositional hardware model checking certification," in *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, A. Nadel and K. Y. Rozier, Eds. IEEE, 2023, pp. 1–11.

[6] N. Froleyks, E. Yu, A. Biere, and K. Heljanko, "Certifying phase abstraction," in *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, ser. Lecture Notes in Computer Science, C. Benzmüller, M. J. H. Heule, and R. A. Schmidt, Eds., vol. 14739. Springer, 2024, pp. 284–303.

[7] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT Competition 2016: Recent developments," in *Proceedings 31st AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 5061–5063.

[8] A. Biere, N. Froleyks, and M. Preiner, 2024. [Online]. Available: https://hwmcc.github.io/2024/

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_6

Efficiently Synthesizing Lowest Cost Rewrite Rules for Instruction Selection

Ross Daly
Stanford University
Stanford, CA, USA
rdaly525@cs.stanford.edu

Caleb Donovick
Stanford University
Stanford, CA, USA
donovick@cs.stanford.edu

Caleb Terrill
Stanford University
Stanford, CA, USA
cdterrill26@gmail.com

Jackson Melchert
Stanford University
Stanford, CA, USA
melchert@stanford.edu

Priyanka Raina
Stanford University
Stanford, CA, USA
praina@stanford.edu

Clark Barrett
Stanford University
Stanford, CA, USA
barrett@cs.stanford.edu

Pat Hanrahan
Stanford University
Stanford, CA, USA
hanrahan@cs.stanford.edu

***Abstract*—Compiling programs to an instruction set architecture (ISA) requires a set of rewrite rules that map patterns consisting of compiler instructions to patterns consisting of ISA instructions. We synthesize such rules by constructing SMT queries, whose solutions represent two functionally equivalent programs. These two programs are interpreted as an instruction selection rewrite rule. Existing work is limited to single-instruction ISA patterns, whereas our solution does not have that restriction. Furthermore, we address inefficiencies of existing work by developing two optimized algorithms. The first only generates unique rules by preventing synthesis of duplicate and composite rules. The second only generates lowest-cost rules by preventing synthesis of higher-cost rules. We evaluate our algorithms on multiple ISAs. Without our optimizations, the vast majority of synthesized rewrite rules are either duplicates, composites, or higher cost. Our optimizations result in synthesis speed-ups of up to 768× and 4004× for the two algorithms.**

I. Introduction

As we approach the end of Moore's law and Dennard scaling, drastically improving computing performance and energy efficiency requires designing domain-specific hardware architectures (DSAs) or adding domain-specific extensions to existing architectures [22]. As a result, many DSAs have been developed in recent years [4], [8], [24], [27], [30], each with its own custom instruction set architecture (ISA) or ISA extension.

Targeting such ISAs from a compiler's intermediate representation (IR) requires a custom library of instruction selection rewrite rules. A rewrite rule is a mapping of an IR pattern to a functionally equivalent ISA pattern. Manual specification of rewrite rules is error-prone, time-consuming, and often incomplete. It is therefore desirable to automatically generate valid rewrite rules.

When specifying instruction selection rewrite rules, there are two common cases. When ISAs have complex instructions, rewrite rules will often map multi-instruction IR patterns to a single ISA instruction. When ISAs have simple instructions, rewrite rules will often map a single IR instruction to a multi-instruction ISA pattern. A rewrite rule generation tool should be able to create rewrite rules for both cases. We call such rewrite rules *many-to-many* rules.

Generating instruction selectors is not a new idea. Most relevant to this work is Gulwani et al. [21] who use a satisfiability modulo theories (SMT) solver to synthesize a loop-free program that is functionally equivalent to a given specification. Their approach is called component-based program synthesis (CBPS), as each synthesized program must include functional components from a given component library. Buchwald et al. [6] use and extend CBPS to efficiently generate multi-instruction loop-free IR programs equivalent to a single ISA instruction program; that is, they solve the many-to-one rewrite rules synthesis problem. However, multi-instruction ISA programs cannot be synthesized.

Both of these algorithms produce many *duplicate* rules, which are removed during a post-processing step. As we show, this adds significant additional cost. Another issue is that CBPS as currently formulated does not incorporate the notion of optimizing for cost. In practice, we often want only the set of lowest-cost rules, making it unnecessary (and expensive) to generate equivalent higher-cost rules.

This paper presents an algorithm for automatically generating a complete set of many-to-many rewrite rules. We address the above issues by preventing the synthesis of both duplicate and high-cost rules at rule generation time, using exclusion techniques. As a further optimization, we generate

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_7

rules in stages and exclude *composite* rules, i.e., rules that can be composed of smaller rules found in previous stages. These ensure we produce a small but complete set of rewrite rules. Compared to previous work, our approach eliminates unnecessary rules and significantly reduces the time required to produce the unique necessary ones.

Our contributions are as follows:

- We define generalized component-based program synthesis (GCBPS) as the task of synthesizing two functionally equivalent programs using two component libraries. We then present an SMT-based synthesis approach inspired by Gulwani et al. to solve it.
- We present an iterative algorithm $genAll$ to generate all unique many-to-many rules up to a given size. We identify a set of equivalence relations for patterns encoded as programs and for rules that map IR programs to ISA programs. We use these relations to enumerate and exclude duplicate rules. Furthermore, we directly exclude composite rewrite rules. These result in up to a 768$\times$ synthesis speed-up.
- We present an algorithm $genAll_{LC}$ which generates only the lowest-cost rules by incorporating a cost metric in addition to excluding duplicate and composite rewrite rules. This results in a synthesis speed-up up to 4004$\times$.

The rest of the paper is organized as follows. Section II discusses instruction selection, existing rule generation methods, SMT, and program synthesis. Section III describes a program synthesis query for generating many-to-many rules. Section IV presents an algorithm for generating only unique rewrite rules and defines duplicates and composites. Section V presents an algorithm for synthesizing only the lowest-cost rules. Section VI evaluates both algorithms, and Section VII discusses limitations and further optimizations.

II. Background and Related Work

A. Instruction Selection

Instruction selection is the task of translating code in the compiler's intermediate representation (IR) to functionally equivalent code for a target ISA. Typically, a library of rewrite rules is used in instruction selection. A rewrite rule is a mapping from an IR pattern consisting of IR instructions to a functionally equivalent ISA pattern consisting of ISA instructions. Such patterns can be expression trees or directed acyclic graphs (DAGs).

Significant work has been devoted to developing rewrite rule tiling algorithms to perform instruction selection [1], [5], [12], [14]–[17], [19], [26], [29]. For each rule in the rule library, a tiling algorithm first finds all fragments from the IR program in which the rule's IR pattern exactly matches that fragment. Then, the instruction selector finds a tiling of these matches that completely covers the basic block and minimizes the total rule cost according to some cost metric.

Simple instruction selectors only handle tree-based IR patterns, which is inefficient for reused computations. Modern instruction selectors like LLVM use DAG-based matching that allows for both richer rules and better tiling. Koes et al. [26] describe a similar near-optimal DAG-based instruction selection algorithm [5]. We want to generate rules that can be used with such modern DAG-based instruction selectors.

B. Generating Instruction Selectors

Generating instruction selectors from instruction semantics has been a topic of research interest [6], [7], [9], [10], [23]. Dias and Ramsey [10] introduce an algorithm for generating rewrite rules based on a declarative specification of the ISA. While this solves part of the many-to-many rule task, their work relies on an existing set of algebraic rewrite rules for synthesizing semantically equivalent rules. Our work uses SMT for the instruction and program semantics. However, incorporating certain kinds of algebraic rewrite rules could be an avenue for future optimizations.

Daly et al. [9] propose a way to synthesize instruction selection rewrite rules from the register-transfer level (RTL) specification of a processor. Their algorithm requires a set of pre-specified IR patterns. In contrast, we can efficiently synthesize rules that consider all possible multi-instruction IR patterns up to a given size. Their approach for synthesizing complex instruction constants and handling floating point types could be combined with the approaches in this paper.

The most relevant to this work is the work by Buchwald et al. [6], which leverages component-based program synthesis to generate rules with multi-instruction IR patterns and single-instruction ISA patterns. In contrast, our work synthesizes rules with both multi-instruction IR patterns and multi-instruction ISA patterns. We additionally prevent the synthesis of duplicate, composite, and high-cost rewrite rules, unlike any of the above approaches.

C. Program Synthesis and Equivalence

We use SMT-based program synthesis to enumerate a complete set of instruction selection rewrite rules. In program synthesis enumeration, it is common to remove equivalent solutions [3]. We use the equivalence relation defined in Section IV-A to determine equivalent rewrite rules. In prior work [2], observational equivalence (i.e., programs with the same semantics) has been used for de-duplication [2], however observational equivalence does not take into account the structure of the program, which is essential for rewrite rule pattern matching.

D. Logical Setting and Notation

We work in the context of many-sorted logic (e.g., [13]), where we assume an infinite set of variables of each sort and the usual notions of terms, formulas, assignments, and interpretations. Terms are denoted using non-boldface symbols (e.g., X). Boldface symbols (e.g., $\mathbf{X}$) are used for sets, tuples, and multisets, whose elements are either terms or other collections of terms. $\mathbf{Y} := (Y_1, ..., Y_N)$ defines a tuple, where $|\mathbf{Y}| = N$ and Y_i refers to the i-th element. $\mathbf{Z} := \{z^n\}$ defines a multiset, where the multiplicity of element z is $n \in \mathbb{N}$. Both ψ and ϕ are used to denote formulas. $\psi(\mathbf{X})$ is a formula whose free variables are a subset of $\mathbf{X}$. We use $\mathcal{M} \models \psi(\mathbf{X})$ to denote the *satisfiability* relation between the interpretation $\mathcal{M}$ and the formula ψ. Assuming $\mathbf{X}$ is a collection of variables, $\mathcal{M}_{\mathbf{X}}$ denotes the *assignment* to those variables induced by $\mathcal{M}$. For an assignment α, we write $\alpha \models \psi(\mathbf{X})$ if $\mathcal{M} \models \psi(\mathbf{X})$ for every interpretation $\mathcal{M}$ such that $\mathcal{M}_{\mathbf{X}} = \alpha$.

E. Component-based Program Synthesis

CBPS is a program synthesis task introduced by Gulwani et al. The inputs to the task are:

- A *specification* $\mathbf{S} := (\mathbf{I}^S, O^S, \phi_{spec}(\mathbf{I}^S, O^S))$ containing a tuple of input variables $\mathbf{I}^S$, a single output variable O^S, and a formula $\phi_{spec}(\mathbf{I}^S, O^S)$ relating the inputs and the output.
- A *library of components* (e.g., instructions) $\mathbf{K}$, where the k-th component $\mathbf{K}_k := (\mathbf{I}_k, O_k, \phi_k(\mathbf{I}_k, O_k))$ consists of a tuple of input variables $\mathbf{I}_k$, a single output variable O_k, and a formula $\phi_k(\mathbf{I}_k, O_k)$ defining the component's semantics.

An example component for an addition instruction is shown below using the theory of bit-vectors, QF_BV, where $BV_{[n]}$ is an n-bit sort and $+_{[n]}$ is addition modulo 2^n.

$$((I_0 : BV_{[16]}, I_1 : BV_{[16]}), O : BV_{[16]}, I_0 +_{[16]} I_1 = O)$$

The task is to synthesize a valid program functionally equivalent to the specification using each component from $\mathbf{K}$ exactly once.

For notational convenience, we group together the set of all inputs and outputs of the components: $\mathbf{W} := \cup_{(\mathbf{I}_k, O_k, \_) \in \mathbf{K}} (O_k \cup (\cup \mathbf{I}_k))$. Gulwani et al. encode the program structure using a *connection constraint*: $\phi_{conn}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W})$. This is a formula representing how the program inputs ($\mathbf{I}^S$) and program output (O^S) are connected via the components. The connections are specified using *location variables* $\mathbf{L}$. We do not go into the details of how location variables encode connections (they are in [21]). It is sufficient for our purposes to know that these are integer variables, and an assignment to them uniquely determines a way of connecting the components together into a program. The *program semantics* ϕ_{prog} are defined as the components' semantics conjoined with the connection constraint:

$$\phi_{prog}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}) := \left(\bigwedge_k \phi_k(\mathbf{I}_k, O_k) \right) \wedge \phi_{conn}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}). \tag{1}$$

They define a verification constraint that holds if a particular program is both well-formed (specified using a well-formedness constraint ψ_{wfp}) and satisfies the specification ϕ_{spec}:

$$\phi_{verif} := \psi_{wfp}(\mathbf{L}) \wedge \forall \mathbf{I}^S, O^S, \mathbf{W}. \; \phi_{prog}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}) \implies \phi_{spec}(\mathbf{I}^S, O^S). \tag{2}$$

A *synthesis formula* ϕ_{synth} existentially quantifies $\mathbf{L}$ in (2):

$$\phi_{synth} := \exists \mathbf{L}. \forall \mathbf{I}^S, O^S, \mathbf{W}. \; \psi_{wfp}(\mathbf{L}) \wedge \left(\phi_{prog}(\mathbf{L}, \mathbf{I}^S, O^S, \mathbf{W}) \implies \phi_{spec}(\mathbf{I}^S, O^S) \right). \tag{3}$$

This formula can be solved using a technique called counter-example guided inductive synthesis (CEGIS). CEGIS solves such exist-forall formulas by iteratively solving a series of quantifier-free queries and is often more efficient than trying to solve the quantified query directly. More details are in [21]. For our purposes, we assume the existence of a CEGIS implementation, *CEGIS*, which takes an instance of ϕ_{synth} and returns a model $\mathcal{M}$ with the property that $\mathcal{M}_{\mathbf{L}} \models \phi_{verif}$, from which a program that is a solution to CBPS can be constructed.

III. Component-based Program Synthesis for Many-to-Many Rules

Given the IR and ISA instruction sets $\mathbf{K}^{IR}$ and $\mathbf{K}^{ISA}$, Buchwald et al. [6] use CBPS to synthesize rewrite rules. They use a single ISA instruction $\mathbf{k}^{ISA} \in \mathbf{K}^{ISA}$ for the CBPS specification and a subset of the IR instructions for the CBPS components. A solution to the resulting ϕ_{synth} formula gives a program $\mathbf{P}^{IR}$. If $\mathbf{P}^{ISA}$ is the single-instruction program consisting of $\mathbf{k}^{ISA}$, they interpret the pair $(\mathbf{P}^{IR}, \mathbf{P}^{ISA})$ as an instruction selection rewrite rule.

However, Buchwald et al.'s solution is insufficient for generating many-to-many rules, as they cannot synthesize IR and ISA programs that both contain multiple instructions. Instead, two functionally equivalent programs need to be synthesized. We first define an extension to CBPS called generalized component-based program synthesis (GCBPS) to address this problem. Then, we show how to construct a synthesis query whose solutions represent pairs of functionally equivalent programs.

A. Generalized Component-based Program Synthesis

We define the GCBPS task as that of synthesizing two programs, $\mathbf{P}^a$ and $\mathbf{P}^b$, represented using location variables $\mathbf{L}^a$ and $\mathbf{L}^b$, given two sets of components $\mathbf{K}^a$ and $\mathbf{K}^b$, two sets of inputs $\mathbf{I}^a, \mathbf{I}^b$ where $|\mathbf{I}^a| = |\mathbf{I}^b|$, and two outputs O^a, O^b where the following conditions hold true:

1) $\mathbf{P}^a$ uses each component in $\mathbf{K}^a$ exactly once.
2) $\mathbf{P}^b$ uses each component in $\mathbf{K}^b$ exactly once.
3) $\mathbf{P}^a$ is functionally equivalent to $\mathbf{P}^b$.

B. Solving GCBPS

We start with the CBPS verification constraint from (2) using components $\mathbf{K}^a$ (and a corresponding set of inputs and outputs $\mathbf{W}^a$), but modify it slightly by introducing variables $(\mathbf{I}^a, O^a)$ that are fresh copies of $(\mathbf{I}^S, O^S)$:

$$\psi_{wfp}(\mathbf{L}^a) \wedge \forall \mathbf{I}^a, O^a, \mathbf{W}^a, \mathbf{I}^S, O^S. \; (\phi^a_{prog}(\mathbf{L}^a, \mathbf{I}^a, O^a, \mathbf{W}^a) \wedge \phi_{spec}(\mathbf{I}^S, O^S)) \implies \left((\wedge_i \; I^a_i = I^S_i) \implies O^a = O^S \right). \tag{4}$$

Assuming the formulas for both the program and the specification, if their inputs are the same, their outputs must also be the same.

We next replace the specification program with a different component-based program using components $\mathbf{K}^b$ and quantify over that program's inputs $\mathbf{I}^b$, output O^b, and component variables $\mathbf{W}^b$:

$$\phi_{verif} := \psi_{wfp}(\mathbf{L}^a) \land \psi_{wfp}(\mathbf{L}^b) \land \forall \mathbf{I}^a, \mathbf{I}^b, O^a, O^b, \mathbf{W}^a, \mathbf{W}^b. \tag{5}$$
$$(\phi^a_{prog}(\mathbf{L}^a, \mathbf{I}^a, O^a, \mathbf{W}^a) \land \phi^b_{prog}(\mathbf{L}^b, \mathbf{I}^b, O^b, \mathbf{W}^b)) \implies$$
$$((\land_i\, I^a_i = I^b_i) \implies O^a = O^b)\,.$$

This is our generalized verification constraint stating the correctness criteria for when two component-based programs are semantically equivalent.

To synthesize such a pair of programs, a synthesis formula ϕ_{synth} is defined by existentially quantifying $\mathbf{L}^a$ and $\mathbf{L}^b$ in the verification formula (5):

$$\phi_{synth} := \exists \mathbf{L}^a, \mathbf{L}^b. \forall \mathbf{I}^a, \mathbf{I}^b, O^a, O^b, \mathbf{W}^a, \mathbf{W}^b. \tag{6}$$
$$\psi_{wfp}(\mathbf{L}^a) \land \psi_{wfp}(\mathbf{L}^b) \land$$
$$\Big((\phi^a_{prog}(\mathbf{L}^a, \mathbf{I}^a, O^a, \mathbf{W}^a) \land \phi^b_{prog}(\mathbf{L}^b, \mathbf{I}^b, O^b, \mathbf{W}^b)) \implies$$
$$((\land_i\; I^a_i = I^b_i) \implies O^a = O^b) \Big).$$

As above, we assume that calling *CEGIS* on ϕ_{synth} returns a model $\mathcal{M}$ such that $\mathcal{M}_{\mathbf{L}^a \cup \mathbf{L}^b} \models \phi_{verif}$. This can be converted into a pair of programs $(\mathbf{P}^a, \mathbf{P}^b)$ representing a rewrite rule that is a solution for the GCBPS task. We write $\mathit{rewriteRule}(\mathbf{K}^a, \mathbf{K}^b, \mathcal{M}_{\mathbf{L}^a}, \mathcal{M}_{\mathbf{L}^b})$ for the rewrite rule constructed from a specific model $\mathcal{M}$ using the component sets $\mathbf{K}^a$ and $\mathbf{K}^b$.

IV. Generating All Many-to-Many Rewrite Rules

Buchwald et al. [6] describe an iterative algorithm, *IterativeCEGIS*, to synthesize rewrite rules using CBPS. This algorithm iterates over all multisets of IR instructions up to a given size and only runs synthesis on each such multiset. Compared to running synthesis using all the IR instructions at once, this iterative algorithm works better in practice.

However, *IterativeCEGIS* cannot synthesize rewrite rules with both multi-instruction IR programs and multi-instruction ISA programs. Furthermore, it produces duplicate rewrite rules which are then filtered out in a post-synthesis filtering step. Although the results are correct, this approach is highly inefficient because each call to CEGIS is expensive, and a CEGIS call is made, not just for some duplicate rules, but for every duplicate rule. In our approach, we make the requirement that a solution is not a duplicate part of the CEGIS query itself, ensuring that each successful CEGIS query finds a new, non-redundant rewrite rule.

Our iterative algorithm, *genAll*, is shown in Figure 1. It takes as parameters the IR and ISA component sets, $\mathbf{K}^{IR}$

```
1  $genAll(\mathbf{K}^{IR}, \mathbf{K}^{ISA}, N^{IR}, N^{ISA})$:
2    $\mathbb{S}_R \leftarrow \{\}$
3    for $n_1, n_2 \in [1, N^{IR}] \times [1, N^{ISA}]$:
4      for $\mathbf{m}^{IR} \in multicomb(\mathbf{K}^{IR}, n_1)$:
5        for $\mathbf{m}^{ISA} \in multicomb(\mathbf{K}^{ISA}, n_2)$:
6          for $\mathbf{I}^{IR}, \mathbf{I}^{ISA} \in allInputs(\mathbf{m}^{IR}, \mathbf{m}^{ISA})$:
7            $\phi, \mathbf{L}^{IR}, \mathbf{L}^{ISA} \leftarrow$
                 $GCBPS(\mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{I}^{IR}, \mathbf{I}^{ISA})$
8            $\phi \leftarrow \phi\ \land \neg AllComposites(\mathbb{S}_R, \ldots)$
9            $\mathbb{S}_R\ \leftarrow \mathbb{S}_R \cup$
                 $CEGISAll(\phi, \mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{L}^{IR}, \mathbf{L}^{ISA})$
10   return $\mathbb{S}_R$
```

Fig. 1: Iterative algorithm to generate all unique rewrite rules up to a given size.

```
1  $CEGISAll(\phi, \mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{L}^{IR}, \mathbf{L}^{ISA})$:
2    $\mathbb{S}_R = \{\}$
3    while True:
4      $\mathcal{M} \leftarrow CEGIS(\phi)$
5      if $\mathcal{M} = \bot$: return $\mathbb{S}_R$
6      $\mathbf{R} \leftarrow rewriteRule(\mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathcal{M}_{\mathbf{L}^{IR}}, \mathcal{M}_{\mathbf{L}^{ISA}})$
7      $\mathbb{S}_R \leftarrow \mathbb{S}_R \cup \{\mathbf{R}\}$
8      $\phi \leftarrow \phi \land \neg\psi_{dup}(\mathbf{R}, (\mathbf{L}^{IR}, \mathbf{L}^{ISA}))$
```

Fig. 2: AllSAT algorithm to synthesize all unique rules. Line 8 excludes all rules that are duplicates of the current synthesized rewrite rule.

and $\mathbf{K}^{ISA}$ respectively, as well as a maximum number of components of each kind to use in rewrite rules, N^{IR} and N^{ISA}, and iteratively builds up a set $\mathbb{S}_R$ of rewrite rules, which it returns at the end. Line 3 shows that n_1 and n_2 iterate up to these maximum sizes. Line 4 iterates over all multisets of elements from $\mathbf{K}^{IR}$ of size n_1 using a standard multicombination algorithm *multicomb* [25] (not shown). Line 5 is similar but for multisets from $\mathbf{K}^{ISA}$ of size n_2. Next, for a given choice of multisets, line 6 enumerates all possible ways of selecting input vectors from those multisets that could create well-formed programs by constructing two fresh sets of input variables. Line 7 constructs fresh sets of location variables $\mathbf{L}^{IR}$ and $\mathbf{L}^{ISA}$ and returns them along with the instantiated GCBPS synthesis formula (using Equation (6)).[1] Line 8 excludes all *composite rules* from the synthesis search space. Composite rules are rules that can be constructed using the current set of rules $\mathbb{S}_R$ and are thus unnecessary for instruction selection. We discuss this in more detail in Section IV-B. Finally, on line 9, the current set of rules $\mathbb{S}_R$ is updated with the result of calling *CEGISAll*, which we describe next.

Figure 2 shows the *CEGISAll* algorithm that performs the AllSAT [20], [31] task. Its parameters are the synthesis formula ϕ, the multisets $\mathbf{m}^{IR}$ and $\mathbf{m}^{ISA}$, and the location variables $\mathbf{L}^{IR}$ and $\mathbf{L}^{ISA}$. It returns a set $\mathbb{S}_R$ of rewrite rules. Initially, this set is empty. The algorithm iteratively calls

[1]We augment the well-formed program constraint in (6) to prevent synthesizing programs containing dead code and unused inputs. This can be accomplished by enforcing that each input and intermediate value is used in at least one location.

a standard $CEGIS$ algorithm to solve the synthesis query, constructing a new rewrite rule $\mathbf{R}$, which is added to the set $\mathbb{S}_R$ of rewrite rules, when the call to $CEGIS$ is successful. The iteration repeats until the $CEGIS$ query returns $\perp$, indicating that there are no more rewrite rules to be found. Note that after each iteration, the ϕ_{synth} formula is refined by adding the negation of a formula capturing the notion of duplicates for this rule. We describe how this is done next.

A. Excluding Duplicate Rules

Consider the two distinct rules below. As a syntactical convention, infix operators are used for IR patterns and function calls for ISA patterns.

$$I_1 + (I_2 \cdot I_3) \to add(I_1, mul(I_2, I_3))$$
$$(I_1 \cdot I_3) + I_2 \to add(I_2, mul(I_1, I_3))$$

The two IR patterns represent the same operation despite the fact that the variable names and the order of the commutative arguments to addition are both different. Both rules would match the same program fragments in an instruction selector and would result in the same rewrite rule application. Thus, we consider such rules to be equivalent and would like to ensure that only one is generated by our algorithm.

We first define a rewrite rule equivalence relation, $\sim_{rule}$. Informally, two rules are equivalent if replacing either one by the other has no discernible effect on the execution of an instruction selection algorithm. We make this more formal by considering various attributes of standard instruction selection algorithms.

Commutative Instructions Modern pattern matching algorithms used for instruction selection try all argument orderings for commutative instructions [5]. We define the commutative equivalence relation $\sim_{C^{IR}}$ as $\mathbf{P}_1^{IR} \sim_{C^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_2^{IR}$ is a remapping of $\mathbf{P}_1^{IR}$'s commutative instruction's arguments.

Same-kind Instructions Programs $\mathbf{P}$ generated by GCPBS have a unique identifier, the program line number, for each instruction. This means that if two instructions of the same kind appear in a program, interchanging their line numbers results in a different program, even though it makes no difference to the instruction selection algorithm. We define the same-kind equivalence relation $\sim_{K^{IR}}$ as $\mathbf{P}_1^{IR} \sim_{K^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_2^{IR}$ is the result of remapping the line numbers for same-kind instructions in $\mathbf{P}_1^{IR}$.

Data Dependency Modern instruction selection algorithms perform pattern matching, not based on a total order of instructions, but on a partial order determined by data dependencies. Many different sequences may thus lead to the same partial order. We define $\sim_{D^{IR}}$ as $\mathbf{P}_1^{IR} \sim_{D^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_1^{IR}$ and $\mathbf{P}_2^{IR}$ have the same data dependency graph.

Rule Input Renaming For a given rewrite rule, the input variables used for the IR program must match the input variables used for the ISA program, but the specific variable identifiers used do not matter. We define the equivalence relation $\sim_{I^{rule}}$ on rules (i.e., pairs of programs) as $\mathbf{R}_1 \sim_{I^{rule}} \mathbf{R}_2$ iff $\mathbf{R}_2$ is the result of remapping variable identifiers in $\mathbf{R}_1$.

Rule Equivalence The first three equivalence relations defined above are for IR programs, but the analogous relations ($\sim_{C^{ISA}}$, $\sim_{K^{ISA}}$, $\sim_{D^{ISA}}$) for ISA instructions are also useful.

Putting everything together, we define rule equivalence $\sim_{rule}$ as follows.

$$\sim_{IR} \;:= \uplus\{\sim_{C^{IR}}, \sim_{K^{IR}}, \sim_{D^{IR}}\} \tag{7}$$
$$\sim_{ISA} \;:= \uplus\{\sim_{C^{ISA}}, \sim_{K^{ISA}}, \sim_{D^{ISA}}\} \tag{8}$$
$$\sim_{rule} \;:= \uplus\{(\sim_{IR} \otimes \sim_{ISA}), \sim_{I^{rule}}\} \tag{9}$$

Overall IR equivalence is defined as the transitive closure of the union (notated with $\uplus$) of the three individual IR relations. ISA equivalence is defined similarly. Overall rewrite rule equivalence is then defined using the $\otimes$ operator, where $\sim_\otimes = \sim_a \otimes \sim_b$ is defined as: $(a_1, b_2) \sim_\otimes (a_2, b_2)$ iff $a_1 \sim_a a_2$ and $b_1 \sim_b b_2$. Specifically, rule equivalence is obtained by combining IR equivalence in this way with ISA equivalence, and then combining the result with $\sim_{I^{rule}}$ using $\uplus$.

The set of all duplicates of rule $\mathbf{R}$ is the rule equivalence class $[\mathbf{R}]_{rule}$, where $\mathbf{R}' \in [\mathbf{R}]_{rule} \iff \mathbf{R} \sim_{rule} \mathbf{R}'$. ψ_{dup} can be constructed as the disjunction of all elements of the equivalence class $[\mathbf{R}]_{rule}$

B. Excluding Composite Rules

We also exclude any rule whose effect can already be achieved using the current set of generated rules (line 8 of Figure 1). We elucidate this using a simple example. Assume the algorithm just constructed a new query for the multisets $\mathbf{m}^{IR}$, $\mathbf{m}^{ISA}$, and the input $\mathbf{I}^{IR}$ (line 7 of Figure 1), and assume that the rule library $\mathbb{S}_R$ currently contains rules for addition ($I_1 + I_2 \to add(I_1, I_2)$) and multiplication ($I_1 \cdot I_2 \to mul(I_1, I_2)$). Consider the following cases.

1) If $\mathbf{I}^{IR} = (I_1)$, $\mathbf{m}^{IR} = \{+\}$, and $\mathbf{m}^{ISA} = \{add\}$, then the rule $I_1 + I_1 \to add(I_1, I_1)$ will be synthesized by *CEGISAll*. But this rule is a *specialization* of the existing rule for addition. Any use of this specialized rule could instead be replaced by the more general rule, and this rule can thus be excluded. Note that we order the inputs on line 6 of Figure 1 to guarantee that the most general version of a rule is found first.
2) If $\mathbf{I}^{IR} = (I_1, I_2, I_3)$, $\mathbf{m}^{IR} = \{+, \cdot\}$, and $\mathbf{m}^{ISA} = \{add, mul\}$, then the composite rule $(I_1 + (I_2 \cdot I_3)) \to add(I_1, mul(I_2, I_3))$ will be synthesized by *CEGISAll*. Using similar logic, any use of this composite rule could instead use the simpler and more general rules for addition and multiplication, and this rule can thus be excluded. The multiset ordering used in lines 4 and 5 of Figure 1 ensures that subsets are visited before supersets, guaranteeing that smaller rules are found first. A specialized rule can be interpreted as a composite rule composed of the general rule with fewer inputs.

Only composite rules that would have been synthesized for a particular query need to be excluded. In general, for a specific query based on $\mathbf{m}^{IR}$, $\mathbf{m}^{ISA}$, and $\mathbf{I}^{IR}$, we enumerate and exclude composite rules $\mathbf{R} := (\mathbf{P}^{IR}, \mathbf{P}^{ISA})$ that meet the following criteria:

```
1  $genAll_{LC}(\mathbf{K}^{IR}, \mathbf{K}^{ISA}, N^{IR}, N^{ISA}, cost)$:
2    $\mathbf{K}_{sorted} \leftarrow sortByCost(\mathbf{K}^{ISA}, N^{ISA}, cost)$
3    $\mathbb{S}_R \leftarrow \{\}$
4    for n $\in [1, N^{IR}]$:
5      for $\mathbf{m}^{IR} \in multicomb(\mathbf{K}^{IR}, n)$:
6        for $\mathbf{m}^{ISA} \in \mathbf{K}_{sorted}$:
7          $c_{cur} \leftarrow cost(\mathbf{m}^{ISA})$
8          for $\mathbf{I}^{IR}, \mathbf{I}^{ISA} \in allInputs(\mathbf{m}^{IR}, \mathbf{m}^{ISA})$:
9            $\phi, \mathbf{L}^{IR}, \mathbf{L}^{ISA} \leftarrow$
                 $GCBPS(\mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{I}^{IR}, \mathbf{I}^{ISA})$
10           $\phi \leftarrow \phi \land \neg AllComposites_{LC}(\mathbb{S}_R, c_{cur}, \ldots)$
11           $\mathbb{S}_R \leftarrow \mathbb{S}_R \cup$
                 $CEGISAll_{LC}(\phi, \mathbf{m}^{IR}, \mathbf{m}^{ISA}, \mathbf{L}^{IR}, \mathbf{L}^{ISA})$
12   return $\mathbb{S}_R$
```

Fig. 3: Iterative algorithm to generate all lowest-cost rules. ISA multisets are ordered by cost. *CEGISAll* is modified to exclude rules with duplicate IR programs.

- $\mathbf{R}$ has exactly $|\mathbf{I}^{IR}|$ inputs.
- $\mathbf{P}^{IR}$ has the same components as $\mathbf{m}^{IR}$.
- $\mathbf{P}^{ISA}$ has the same components as $\mathbf{m}^{ISA}$.
- $\mathbf{P}^{IR}$ is built from the IR programs of already-found rules in $\mathbb{S}_R$.
- $\mathbf{P}^{ISA}$ is the result of applying the rewrite rules used to build $\mathbf{P}^{IR}$.

This enumeration is encapsulated by the call to *AllComposites* on line 8 of Figure 1.

V. Generating All Lowest-Cost Rules

Because all duplicates are excluded, the *genAll* algorithm generates only unique rewrite rules. However, two unique rules can share the same IR pattern. For a particular IR pattern, only the lowest-cost rule is needed for some cost metric. Knowing the instruction selection cost metric at rule-generation time presents another time-saving opportunity because we can also prevent the synthesis of high-cost rules.

We make a few assumptions about such a cost metric.

- The cost for an instruction selection tiling is equal to the sum of the costs of each tiling rule's ISA program.
- The cost of an ISA program $\mathbf{P}^{ISA}$ only depends on the instruction *contents*, not the program structure. This cost is the sum of the cost of each instruction in the program.

While these assumptions are a restriction on the space of possible cost metrics, they are sufficient to represent common ones like code size and energy. If the compiler's cost metric violates these assumptions, the *genAll* algorithm can be used instead. This restricted space of cost metrics has the important property that the cost of any rule that would be synthesized using the components $\mathbf{m}^{ISA}$ can be determined up front as the sum of the cost of each component.

Figure 3 shows our synthesis algorithm updated to only synthesize the lowest-cost rules for each unique IR pattern. The first change is to sort all possible mulitsets of ISA instructions up to size N^{ISA} by cost (lower cost first) (line 2). This ordering ensures that the first rule synthesized for a particular IR program will be the lowest-cost version of that rule. Therefore, after synthesizing a new rule, all rules with a duplicate IR program can be excluded. The second change excludes rules with duplicate IR programs. A duplicate IR program is defined using the IR equivalence relation:

$$\sim_{IR_{LC}} := \uplus\{\sim_{C^{IR}}, \sim_{K^{IR}}, \sim_{D^{IR}}, \sim_{I^{IR}}\} \qquad (10)$$

This is the same definition as (7), but with an additional relation $\sim_{I^{IR}}$ defined as $\mathbf{P}_1^{IR} \sim_{I^{IR}} \mathbf{P}_2^{IR}$ iff $\mathbf{P}_2^{IR}$ is the result of remapping variable identifiers in $\mathbf{P}_1^{IR}$. The $CEGISAll_{LC}$ function called on line 11 is the same as *CEGISAll*, except that it uses $\sim_{IR_{LC}}$ instead of $\sim_{IR}$ when constructing ψ_{dup}.

The third change modifies *AllComposites* to use the known up-front cost $cost(\mathbf{m}^{ISA})$. To see how this works, we consider again the example from Section IV-B. As before, we assume $\mathbb{S}_R$ currently contains two rules: one for addition $(I_1 + I_2 \rightarrow add(I_1, I_2))$ and one for multiplication $(I_1 \cdot I_2 \rightarrow mul(I_1, I_2))$. We assume the target (ISA) expressions for these rules have cost 5 and 10, respectively. Consider the following situation:

- Suppose $\mathbf{I}^{IR} = (I_1, I_2, I_3)$, and $\mathbf{m}^{IR} = \{+, \cdot\}$. It might be possible to synthesize a rule that has IR pattern $(I_1 + (I_2 \cdot I_3))$. We know that the composite rule $(I_1 + (I_2 \cdot I_3)) \rightarrow add(I_1, mul(I_2, I_3))$ would have a cost of 15 since rule costs are additive. Therefore, we can exclude any rule that matches this IR pattern and has $cost(\mathbf{m}^{ISA}) \geq 15$.

To implement this, only one adjustment needs to be made to the conditions in Section IV-B. Instead of requiring $\mathbf{P}^{ISA}$ to have the same components as $\mathbf{m}^{ISA}$, we simply require $cost(\mathbf{P}^{ISA}) \geq cost(\mathbf{m}^{ISA})$, i.e., for rules matching the other conditions, if the ISA program has a cost equal to or greater than cost of the ISA program in the current rule, it is excluded. These conditions are encapsulated by the call to $AllComposites_{LC}$ (line 10).

VI. Evaluation

Our evaluation strategy is threefold. We first show that our algorithm is capable of producing a variety of many-to-many rules. A good set of rewrite rules involves both many-to-one and one-to-many rules. We also show that by removing duplicate, composite, and high-cost rules, we produce a much smaller set of rewrite rules. Second, we analyze the effect on performance of the optimizations described above. We show that they all significantly reduce the time spent in synthesis. Finally, we show that by using different cost metrics, we can generate different sets of lowest-cost rewrite rules.

A. Implementation

All instructions are formally specified using the hwtypes Python library [11], which leverages pySMT [18] to construct (quantifier-free) SMT queries in the theory of bit-vectors. We also use annotations indicating which instructions are commutative. We use Boolector [28] as the SMT solver and set a timeout of 12 seconds for each CEGIS invocation. Every synthesized rewrite rule is independently verified to be valid.

B. Instruction Specifications

To evaluate our algorithms, we selected small but non-trivial sets of IR and ISA instructions operating on 4-bit bit-vectors.

IR We define the IR instruction set to be constants (0, 1), bitwise operations (not, and, or, xor), arithmetic operations (neg, add, sub), multiplication (mul), unsigned comparison operations (ult, ule, ugt, uge), equality (eq), and dis-equality (neq).

ISA 1 This is a minimal RISC-like ISA containing only 6 instructions: $nand$, sub, three comparison instructions ($cmpZ$, $cmpN$, $cmpC$) which compute the zero (Z), sign (N), and carry (C) flags respectively for a subtraction, and a flag inverting instruction (inv).

ISA 2 This is an ISA specialized for linear algebra. It supports the 5 instructions: neg, add, $add3$ (addition of 3 values), mul, and mac (multiply-accumulate).

C. Rewrite Rule Synthesis

For each ISA we run three experiments. The first experiment (All Rules) is the baseline that generates all many-to-many rules, including duplicate, composite, and high cost rules. This is an implementation of Buchwald et al.'s *IterativeCEGIS* algorithm extended to use GCBPS for many-to-many rules (notated as *IterativeCEGIS*$_{GCBPS}$). The second (Only Unique) generates only unique rules by excluding all duplicates and composites using the *genAll* algorithm. The third (Only Lowest-Cost) generates only the lowest-cost rules using the *genAll*$_{LC}$ algorithm in Figure 3. A code-size cost metric is used, i.e., *cost*($\mathbf{K}$) is just the number of components in $\mathbf{K}$.

For ISA 1, we split the rule generation into two parts. The first part (ISA 1a) synthesizes rules composed of bitwise and arithmetic IR instructions using the ISA's $nand$ and sub instructions. The second part (ISA 1b) synthesizes rules composed of constants and comparison instructions using the four instructions $cmpZ$, $cmpN$, $cmpC$, and inv.

For 1a and 1b, we synthesize rewrite rules up to an IR program size of 2 and an ISA program size of 3 (written 2-to-3). For (Only Lowest-Cost), we increase the ISA program size to 5 and 4 respectively. For ISA 2, we synthesize all rewrite rules composed of constant, and arithmetic (including mul) IR instructions up to size 3-to-2.

The number of rewrite rules produced for each configuration for ISA 1a, 1b, and 2 is shown in Tables I, II, and III, respectively. Each table entry is the number of rewrite rules synthesized for a particular IR and ISA program size. For all ISAs, the extra synthesized rules in (All Rules) were compared against the duplicate and composite rules excluded by (Only Unique). Entries in (All Rules) marked with a '(-n)' represent 'n' rules that (Only Unique) synthesized, but (All Rules) missed due to CEGIS timeouts. The (All Rules) experiment for the entry marked with an asterisk could not complete in 70 hours, so the number calculated from (Only Unique) is shown.

For both ISAs we were able to synthesize 1-to-many and many-to-1 rules for both IR and ISA instructions. *genAll* produced a more complete set of rules than *IterativeCEGIS*$_{GCBPS}$.

Table IV shows the percentage of rules that are duplicates or composites in the first column, and the percentage of rules that are high cost in the second column. Most rules in (All Rules) are duplicates, composites, or high cost. Out of the 349179 rules up to size 3-to-2 for ISA 2 (i.e., the sum of the (All Rules)), 99.5% are duplicates or composites. Similarly, most rules are high cost. In ISA 1a, 59672 out of 59822 rules (99.7%) up to size 2-to-3 are high cost.

D. Synthesis Time Improvement with genAll

In this section we showcase the synthesis time improvements of *genAll*. The first experiment is the baseline *IterativeCEGIS*$_{GCBPS}$. The second excludes duplicate rules (i.e., with line 8 of Figure 2). The third, *genAll*, excludes both duplicates and composites (i.e. with line 8 of both Figure 2 and Figure 1).

For each *GCBPS* query, we note the time required (t_{sat}) to run *CEGISAll*. Next, we measure the number of unique rules (N_{unique}) found by *CEGISAll*. We then add the pair (N_{unique}, t_{sat}) to our dataset. We plot the cumulative synthesis time versus the number of unique rules found by doing the following. Each data point is sorted by its slope (t_{sat}/N_{unique}). Then, the increase in both t_{sat} and N_{unique} is plotted for each sorted point. Some data points have $N_{unique} = 0$ indicating that every synthesized rule was redundant and is shown using a vertical slope.

The synthesis time plot for unique rewrite rules for ISA 1b up to size 2-to-3 is shown in Figure 4a. Excluding all duplicates shows a 5.3× speedup. Excluding both duplicates and composites shows a 6.2× speedup. Both optimizations find an additional 5 unique rules.

E. Synthesis Time Improvement with genAll$_{LC}$

We also showcase the synthesis time improvements of *genAll*$_{LC}$ using a similar setup. The first experiment is the baseline *IterativeCEGIS*$_{GCBPS}$. The second excludes IR duplicate rules. The third, *genAll*$_{LC}$, excludes both IR duplicates and IR composites.

We use the same experimental setup as before, except when computing N_{unique}, all higher-cost rules are filtered instead. The synthesis time plot for lowest-cost rewrite rules for ISA 1b up to size 2-to-3 is shown in Figure 4b.

Excluding rules with duplicate IR programs provides a 41× speed-up. Also excluding high-cost composites provides a 1254× speed-up over the baseline (All Rules) configuration.

F. Total Speed-up

We summarize the speed-ups of *genAll* and *genAll*$_{LC}$ compared to the *IterativeCEGIS*$_{GCBPS}$ baseline for all configurations in Table V. We compare the synthesis time in the "Synth" column. We compare the total algorithm runtime in the "Total" column (including time for iterating, solving, rule filtering, etc.). The last row's baseline did not complete in 70 hours, so we provide lower bounds for speed-up.

		ISA Program Size										
		All Rules			Only Unique			Only Lowest-Cost				
		1	2	3	1	2	3	1	2	3	4	5
IR Prog	1	5	32	1096	3	10	96	3	4	2	1	0
Size	2	76	1719	56894	40	189	1940	40	67	34	12	6

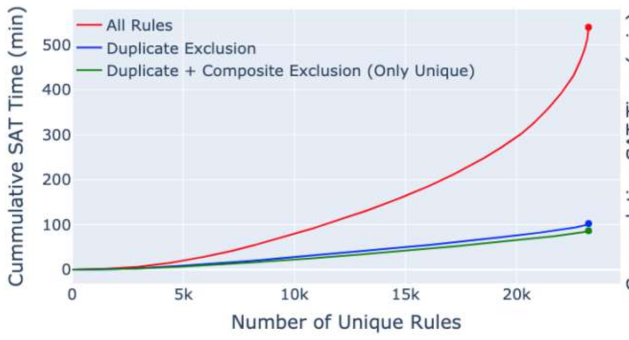
TABLE I: Number of synthesized rewrite rules for ISA 1a.

		ISA Program Size									
		All Rules			Only Unique			Only Lowest-Cost			
		1	2	3	1	2	3	1	2	3	4
IR Program	1	17	71	3662	9	51	873	7	3	0	0
Size	2	89	3942 (-5)	199572	78	717	21511	52	64	9	0

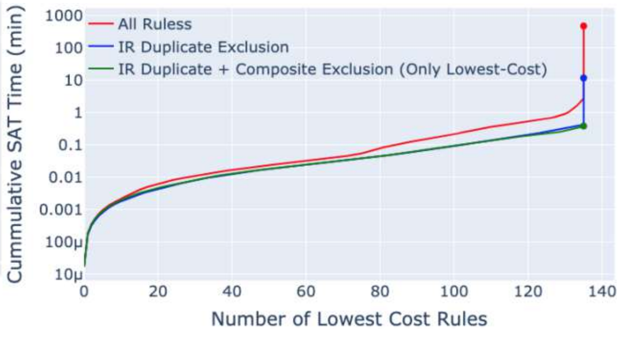
TABLE II: Number of synthesized rewrite rules for ISA 1b.

		IR Program Size								
		All Rules			Only Unique			Only Lowest-Cost		
		1	2	3	1	2	3	1	2	3
ISA Program	1	11	287	3998	3	14	315	3	14	315
Size	2	10	3115	341758*	3	69	1337	1	32	760

TABLE III: Number of synthesized rewrite rules for ISA 2.



(a) $genAll$



(b) $genAll_{LC}$

Fig. 4: Cumulative synthesis time comparison for ISA 1b up to size 2-to-3.

ISA	Rule Size up to (IR, ISA)	% Duplicate or Composite	% High-cost
1a	(2, 3)	96.2%	99.7%
1b	(2, 3)	88.8%	99.9%
2	(3, 2)	99.5%	99.7%

TABLE IV: Percent of rewrite rules up to (IR, ISA) size that are a duplicate or a composite, and percent that are high-cost.

The speed-ups depend on many parameters including the maximum size of the rewrite rules, the number of possible instructions, the commutativity of the instructions, and the semantics of the instructions. The optimizations discussed produce several orders of magnitude speed-ups. Further optimizing the non-solver portions (e.g., re-coding in C) would drastically increase the “Total” speed-ups to be closer to the “Synth” ones. Clearly, the combination of all optimizations discussed in this paper can produce speed-ups of several orders of magnitude.

G. Cost Metric Comparisons

Our final experiment explores how the choice of cost metric influences the rules. We have implemented two cost metrics: a code size metric (CS) and an estimated energy metric (E).

ISA	Rule Size up to (IR, ISA)	$genAll$ Speed-up		$genAll_{LC}$ Speed-up	
		Synth	Total	Synth	Total
1a	(2, 2)	3.5×	1.3×	11×	2.8×
1b	(2, 2)	3.1×	1.7×	26×	2.8×
2	(2, 2)	11×	2×	53×	2.5×
1a	(2, 3)	12×	6.8×	601×	57×
1b	(2, 3)	6.2×	2.7×	1254×	63×
2	(3, 2)	$>$ 768×	$>$ 81×	$>$ 4004×	$>$ 171×

TABLE V: Speed-ups compared to $IterativeCEGIS_{GCBPS}$.

ISA	Rule Size up to (IR, ISA)	Unique (CS)	Unique (E)	Common
1a	(2, 5)	121	161	48
1b	(2, 4)	99	198	36
2	(3, 2)	134	137	991

TABLE VI: Number of unique and common rewrite rules synthesized for code size (CS) and energy (E) cost metrics.

The energy metric was created to correspond to real hardware energy data. For example the cost ratio for mul and add is $1 : 1$ for code size, but is $2.5 : 1$ for energy. The number of common and unique lowest-cost rewrite rules for each ISA is shown in Table VI.

While there is some overlap in common rules, each cost metric produces a differing set of unique lowest-cost rules.

VII. Conclusion and Future Work

We showed that many-to-many instruction selection rewrite rules can be synthesized for various ISAs using program synthesis. This supports two major trends in computer architecture. The first is the trend towards simple or reduced instruction architectures where multiple instructions are needed for simple operations. It also supports the trend to introduce more complex domain-specific instructions for energy efficiency. In this case, a single instruction can implement complex operations.

We showed that our algorithms are efficient. Removing duplicates, composites, and higher-cost rules results in multiple orders of magnitude speed-ups. Synthesizing many-to-many rewrite rules for modern IRs and ISAs may require further optimizations. Many of our synthesized rules contain program fragments that a compiler would optimize during IR optimization or peephole optimization. A modified version of GCBPS could be used to directly synthesize and exclude such program fragments.

Buchwald et al. [6] presented generalizations for multi-sorted instructions, multiple outputs, preconditions, and internal attributes, enabling the modeling of memory and control flow instructions. Our synthesis query and algorithms are orthogonal and could incorporate these features, allowing for a broader range of possible instruction sets.

As is the case in prior work, we limit synthesis to loop free patterns. Relaxing this constraint and using other instruction selection algorithms would be an interesting research avenue.

Another promising research direction involves exploring the trade-offs between synthesis time, compile time, and code quality. This could be done by varying the maximum size of rewrite rules, changing the instruction selection algorithms, relaxing the completeness guarantee, or incorporating IR or peephole optimizations.

We believe this research area is fertile ground and hope our work inspires and enables future research endeavors towards the goal of automatically generating compilers for emerging domain-specific architectures.

References

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 934–950. Springer, 2013.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

[4] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, et al. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[5] Eli Bendersky. *A deeper look into the LLVM code generator, Part 1*, Feb 2013.

[6] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.

[7] R. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):173–190, 1980.

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[9] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Synthesizing instruction selection rewrite rules from RTL using SMT. In *Conference on Formal Methods in Computer-Aided Design (FMCAD)*, page 139, 2022.

[10] Joao Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. *ACM Sigplan Notices*, 45(1):403–416, 2010.

[11] Caleb Donovick, Ross Daly, Jackson Melchert, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. Peak: A single source of truth for hardware design and verification. *arXiv preprint arXiv:2308.13106*, 2023.

[12] Helmut Emmelmann, F.-W. Schröer, and Rudolf Landwehr. BEG: A generator for efficient back ends. *ACM Sigplan Notices*, 24(7):227–237, 1989.

[13] Herbert Enderton and Herbert B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[14] Christopher W. Fraser and David R. Hanson. *A retargetable C compiler: Design and implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.

[16] Mahadevan Ganapathi. *Retargetable code generation and optimization using attribute grammars*. PhD thesis, 1980. AAI8107834.

[17] Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 108–119, New York, NY, USA, 1982. Association for Computing Machinery.

[18] Marco Gario and Andrea Micheli. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*, 2015.

[19] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 231–254, New York, NY, USA, 1978. Association for Computing Machinery.

[20] Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5*, pages 275–289. Springer, 2004.

[21] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[22] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[23] Roger Hoover and Kenneth Zadeck. Generating machine specific optimizing compilers. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–229, 1996.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9):50–59, 2018.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005.

[26] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on DAGs. In *Proceedings of the 6th Annual IEEE/ACM*

International Symposium on Code Generation and Optimization, pages 45–54, 2008.

[27] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A Horowitz, Pat Hanrahan, and Priyanka Raina. APEX: A framework for automated processing element design space exploration using frequent subgraph analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 33–45, 2023.

[28] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[29] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application BURS theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 294–308, 1988.

[30] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. *ACM SIGARCH Computer Architecture News*, 45(2):389–402, 2017.

[31] Takahisa Toda and Takehide Soh. Implementing efficient all solutions sat solvers. *Journal of Experimental Algorithmics (JEA)*, 21:1–44, 2016.

Formal Methods in Computer-Aided Design 2024

Extending DRAT to SMT

S Hitarth *¶, Cayden Codel †, Hanna Lachnitt ‡, and Bruno Dutertre §
*Hong Kong University of Science and Technology, Hong Kong
¶IMDEA Software Institute, Madrid, Spain
Email: hitarth.singh@connect.ust.hk
†Carnegie Mellon University, Pittsburgh, PA, USA
Email: ccodel@cs.cmu.edu
‡Stanford University, Stanford, CA, USA
Email: lachnitt@stanford.edu
§Amazon Web Services, Santa Clara, CA, USA
Email: dutebrun@amazon.com

***Abstract*—The soundness of Satisfiability Modulo Theories (SMT) solvers is critical in many applications. One way to ensure soundness is to have solvers generate proofs that can be independently verified. Unfortunately, generating proofs can have a significant overhead. We propose a new proof format (*e*DRAT) that extends the well-known DRAT format from SAT to SMT. *e*DRAT proofs can be generated with little overhead and can be verified by combining existing tools for propositional reasoning with specialized theory checkers. We instrument the CVC5 solver to generate *e*DRAT proofs and we develop checkers for two SMT theories. Our checkers include an untrusted elaborator written in Rust and a formally verified component written in Lean that validates results from the elaborator. Empirical evaluation shows that *e*DRAT has a much lower proof generation overhead than other formats supported by CVC5, and it has comparable or better proof checking times.**

I. Introduction

Satisfiability Modulo Theories (SMT) solvers are used as back-ends in a variety of applications including software verification and testing [24], [18], [19], the verification of distributed systems [26], model checking [20], [9], [8], and security policy analysis [2], [29]. The soundness of SMT solvers is critical for these applications, especially because SMT solvers have become increasingly complex over the years and are therefore subject to bugs.

When an input formula is satisfiable, solvers can produce a model that can generally be checked, but this approach is not applicable when the SMT solver says that the input formula is unsatisfiable. To increase trust in the unsat case, the SMT community has developed solvers that generate a proof that can be independently validated by a trusted checker. Solvers such as CVC5 [3], OpenSMT [23], SMTInterpol [10], veriT [7] and Z3 [12] can produce proofs, for at least some of the logical theories they support. Some have had proof support for many years.

Several proof formats have been proposed for SMT [33], [30], [22], but none has emerged as a standard. One limitation of these formats is that they require fine-grained proofs with small inference steps. While this simplifies proof checking, generating such detailed proofs is expensive and slows down solvers, and the resulting proofs can be very large and slow to validate.

To address these concerns, we propose *extended DRAT* (*e*DRAT), a new SMT proof format that extends DRAT [21], a standard proof format for Boolean satisfiability. Proofs in *e*DRAT are coarse-grained and clausal. They include Boolean resolution steps (as in DRAT and its predecessor DRUP) and SMT-specific clauses called *theory lemmas*.

Along with *e*DRAT, we present VALIDO, a modular and extensible toolchain for checking *e*DRAT proofs. Proof checking with VALIDO is a two-step process. First, we validate the propositional part of the proof with `DRAT-trim` [34] to extract an unsat core. Second, we check that all the theory lemmas in the unsat core are valid using theory-specific checkers. Currently, VALIDO supports two SMT theories: QF_LRA and QF_UF. The VALIDO theory checkers for these two theories have two components:

- An *elaborator* does most of the heavy lifting. It validates theory lemmas using theory-specific decision procedures and generates an *unsatisfiability certificate* for the negation of each lemma.
- A *certificate validator* checks the unsatisfiability certificates produced by the elaborator.

The validator is the only trusted component, as the validity of a certificate is enough to ensure that a lemma is valid, irrespective of how the certificate was generated. To achieve a high degree of confidence in the correctness of our toolchain, we use the Lean theorem prover [14] to develop and prove the soundness of our validators.

Because theory lemmas are individually validated, we can precisely identify incorrect lemmas when proof validation fails. This can aid debugging and guide the search for a minimal counterexample.

We have instrumented the CVC5 solver to generate *e*DRAT proofs, and we have evaluated VALIDO on SMT-LIB benchmarks. Empirical results show that *e*DRAT proof generation has low overhead (less than 10%), as opposed to between 2x and 17x for two other formats supported by CVC5. *e*DRAT proofs are generally smaller, and proof checking time is comparable to or better than with the alternative proof formats.

Our toolchain can generate eDRAT proofs for any theory that cvc5 supports, including theories with quantifiers and theory

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_8

combinations. Our current proof-checking pipeline, VALIDO, is only implemented for QF_UF and QF_LRA. However, validating a theory lemma boils down to solving a (small) SMT problem. For instance, we could leverage CVC5 itself (or some other proof-producing solver) as an elaborator and the associated ALF/LFSC proof checker as the validator. Therefore, our approach is quite general and not limited to simple theories.

A limitation of *e*DRAT is that it requires problem instances to be in conjunctive normal form (CNF), while SMT problems can be arbitrary formulas. The preprocessing, simplification, and rewriting steps that SMT solvers perform to convert formulas to CNF are not expressible in *e*DRAT. Complementary proof techniques are required for checking that the conversion steps the SMT solver used were sound. We discuss possible approaches to bridge this gap.

II. BACKGROUND

SMT is the problem of deciding the satisfiability of formulas in some (typically first-order) logical theory [13], [6]. The mainstream method employed by SMT solvers is conflict-driven clause learning modulo theories (CDCL(T)). This combines the CDCL algorithm from SAT [25] with theory-specific reasoning implemented by theory solvers. Given a formula ϕ in a theory T, the SMT solver first creates a Boolean abstraction ϕ_{abs} of this formula. The abstraction process replaces atoms in the background theory T with Boolean variables. The CDCL(T) algorithm then alternates between Boolean search and theory reasoning. The CDCL solver enumerates (possibly partial) models σ of ϕ_{abs} that are interpreted as conjunctions of literals in theory T. The theory solver checks whether this conjunction is satisfiable in T. If it is, Boolean search can continue and try to extend the assignment to a full model of ϕ_{abs}. If the conjunction of literals is not satisfiable, the theory solver produces a *theory lemma* that is added to the sets of clauses in the CDCL solver. This clause must be inconsistent with σ and will cause the CDCL solver to backtrack.

Modern SMT solvers extend this basic scheme in many ways—for example, with the dynamic creation of new variables and atoms on the fly and with mechanisms such as theory propagation—but the general principle remains. In one direction, the CDCL solver sends candidate Boolean assignments to the theory solver. In the other direction, the theory solver sends new clauses—that is, theory lemmas—to the CDCL solver. As in SAT, an SMT formula is unsatisfiable if the empty clause is derived by this process.

Generating proofs for CDCL(T) solvers is an active area of research, and several proof formats have been proposed. Proofs may include reasoning steps used during preprocessing and simplification of the original formula, conversion to clauses, Boolean resolution in CDCL, and theory-specific reasoning for justifying theory lemmas. Notable proof formats include Alethe [30] (supported by VeriT [7] and CVC5 [3]) and LFSC [33] (supported by CVC5 and ts predecessors CVC4 and CVC3). Both Alethe and LFSC have dedicated checkers [1], [32]. Other proof-producing SMT solvers [12], [10], [23] use solver-specific proof formats [11], [22], [27] and do not use independent checkers.

Most of these formats represent proofs as terms in a proof calculus. Such terms describe a traditional proof tree (or DAG) with the empty clause at the root. Each node in the tree represents a step that derives a conclusion (stored in the node) from the child nodes using a rule of proof calculus. Leaves represent axioms or assumptions (e.g., assertions from the original formula). One can distinguish generic logical frameworks such as LFSC that encode a particular proof calculus, and formats such as Alethe that come with a fixed calculus for a fixed set of theories. Logical frameworks are more flexible, since proof rules can be added to cover new theories and reasoning steps, but not all rules employed by SMT solvers can be compactly encoded in a logical framework. Most solvers use a fixed proof calculus and a dedicated proof format, which is similar to what Alethe provides. The recently introduced AletheLF format (ALF) is a logical framework that relies on an SMT-like syntax (similar to Alethe) in which SMT constructs can be more easily represented. ALF is supported by CVC5-1.1.0 and newer releases.

Coverage and proof granularity vary across solvers. Some solvers, such as CVC5, can generate low-level, high-detail proofs for almost all of the theories they support.[1] Other solvers, such as Z3, support proofs for a subset of theories and use more coarse-grained proofs. Detailed proofs with small inference steps are easier to verify, but generating such proofs can be costly and can introduce significant overhead both in runtime and memory usage.

We propose a coarse proof format that records the clauses produced (and deleted) during execution of the CDCL(T) algorithm. This extends the DRAT format used by Boolean SAT solvers, which is known to have low overhead.

III. DRAT EXTENSIONS FOR SMT

Our new proof format, *e*DRAT records the reasoning steps performed during the execution of the CDCL(T) algorithm. We do not attempt to express preprocessing, simplification, or conversion of a formula to CNF. Instead, *e*DRAT focuses on capturing the theory reasoning (the theory lemmas) and Boolean reasoning steps (the resolution clauses) that the SMT solver generates. We extend DRAT with syntax for defining theory terms and atoms, describing how these atoms map to Boolean variables, and distinguishing between the different types of clauses involved in SMT. We distinguish between three types of clauses: *assertions*, which are clauses from the CNF representation of the original formula; *theory lemmas* generated by the theory solver; and *regular clauses* generated by the CDCL solver.

An *e*DRAT proof consists of four components:

- The definitions of literals and terms used in the problem.
- The input problem converted to CNF.
- The theory lemmas that were emitted by the theory solver during the course of solving.

[1]As far as we know, CVC5 can generate proofs for all theories that do not involve floating points.

- The DRAT proof of the unsatisfiability of the formula as produced by the underlying SAT solver.

A. Syntax of *e*DRAT

SMT-LIB [5] is the standard input format for SMT solvers. The *e*DRAT syntax for term and atom definition is similar to SMT-LIB with a few extensions. Sort and term declarations in *e*DRAT use the following SMT-LIB syntax:

```
(declare-sort <name> <arity>)
(declare-fun <name> ( <sort>* ) <sort> )
```

The *e*DRAT syntax for datatypes and terms is also the same as in SMT-LIB. We introduce two new commands to give names to terms and to map DIMACS variables to literals:

```
(define-let <term-name> <smt-term>)
(define-literal <varid> <atom-name>)
```

The `define-let` command assigns a name to a term. It is a variant of the SMT-LIB `define-fun` command that omits the type of the term. The `define-literal` command states that a Boolean variable is mapped to a given atom. As in DIMACS, boolean variables are represented by positive integers. To simplify processing, the atom is specified by its name, which must appear either as a declaration or in a previous `define-let` command.

As in DRAT, a clause is represented by a list of non-zero literals terminated by 0. A positive integer denotes a positive literal, and a negative integer denotes its negation. The syntax for clausal reasoning is as follows:

```
a <list-of-integers> 0          Input problem clause
t <list-of-integers> 0        Theory reasoning clause
<list-of-integers> 0         Boolean reasoning clause
d <list-of-integers> 0               Clause deletion
```

*e*DRAT adds then two new prefixes to the DRAT syntax: one for assertions, and one for theory lemmas.

Example III.1. The following small example illustrates the *e*DRAT syntax.

```
1  (declare-sort T 0)
2  (declare-sort S 0)
3  (declare-fun f (T) S)
4  (declare-fun y () T)
5  (declare-fun x () T)
6  (define-let aux!312 (f y))
7  (define-let aux!338 (= (f x) (aux!312)))
8  (define-let aux!339 (= x y))
9  (define-literal 1 aux!338)
10 (define-literal 2 aux!339)
11 a 2 0
12 a -1 0
13 t 1 -2 0
14 0
```

This proof tells us that literals `1` and `2` are mapped to the atoms $f(x) = f(y)$ and $x = y$, respectively, where $f : T \to S$ is an uninterpreted function. Lines 11 and 12 state two assertions $x = y$ and $\neg f(x) = f(y)$, respectively, that come from the input problem. Line 13 is a theory lemma: $f(x) = f(y) \vee x \neq y$. The final step is the empty clause, which is derived by Boolean resolution of the three preceding clauses. This shows that the formula $x = y \wedge f(x) \neq f(y)$ is unsatisfiable.

B. Valido: A Toolchain for Checking eDRAT Proofs

*e*DRAT proof checking consists of two separate tasks: checking that the Boolean reasoning steps derive the empty clause, assuming that the theory lemmas are valid; and checking the validity of the theory lemmas. VALIDO is our toolchain for performing these two tasks.

To check the propositional part of the proof, VALIDO constructs a CNF formula ϕ with the input clauses (those with prefix `a`) and the theory lemmas (those with prefix `t`). It then extracts the DRAT proof π from the *e*DRAT file. This proof includes all the clauses corresponding to Boolean resolutions and all the clause deletions, keeping them in the same order as they occur in *e*DRAT. We then use a restricted version of the `DRAT-trim` tool to check that π is valid for ϕ. In this step, we treat all the theory lemmas as axioms and add them to the input clauses.

We restrict `DRAT-trim` to allow only clause additions that satisfy the reverse-unit-propagation (RUP) property and not the more general resolution-asymmetric-tautology (RAT) property, because accepting RAT clauses is not sound for SMT. It is possible for a formula ϕ to be satisfiable in the background theory T and for a clause C to be RAT with respect to ϕ, but for $\phi \wedge C$ to be unsatisfiable in T. This occurs because the addition of RAT clauses may eliminate some of the Boolean models for ϕ. RUP is sound for SMT as adding a RUP clause preserves all the satisfying models of the original formula.

Example III.2. Consider the formula $\phi := (\neg p \vee q) \wedge (p \vee r) \wedge (q \vee \neg r)$ where the propositional variables represent real arithmetic theory terms defined as follows: $p := x < 0$, $q := x \geq 1$, and $r := x \geq 2$. It follows from the definition that the unit clause p is RAT with respect to ϕ.

We can hence conclude that in propositional logic, ϕ is equisatisfiable to $\phi \wedge p$. However, in SMT, it can be readily checked that while the original formula ϕ is satisfiable with a model $x \to 2$, $\phi \wedge p$ is unsatisfiable.

When `DRAT-trim` successfully validates a DRAT proof, it also returns a set of clauses $U \subseteq \phi$ that forms the *unsat core* of the DRAT proof. To check the rest of the *e*DRAT proof, we only need to check the validity of the *core theory lemmas* that appear in the unsat core U.

One way of validating a theory lemma t is to employ existing SMT technology to generate a proof of the unsatisfiability of $\neg t$ in another proof format, and then to check it with a corresponding proof checker. This works, but we pursue a different approach to provide a higher degree of assurance: relying on purpose-built checkers that are provably sound.

In VALIDO, the theory lemmas are checked by two complementary tools that we call the *elaborator* and the *validator*. These tools are instantiated for each background theory T separately.

- An elaborator checks the unsatisfiability of a conjunctive formula in theory T and generates a *proof certificate*.
- A validator is a provably correct tool that takes a proof certificate and a theory lemma as input, and checks that the proof certificate validates the theory lemma.

Algorithm 1 gives an overview of this method. The architecture allows us to write an efficient but untrusted elaborator that generates a proof certificate for every theory lemma in the core. The validator is a simpler component that we develop and prove correct within the Lean 4 theorem prover [14].

Algorithm 1 General Method for Checking *e*DRAT Proofs

```
Input: An eDRAT Proof
Output: Result of eDRAT proof validation
1: (input,                 ▷ Input problem in DIMACS format
   t_lemmas,               ▷ Theory Lemmas in DIMACS format
   drat_proof,             ▷ Boolean Reasoning as DRAT Proof
   definitions) ← Decompose(eDRAT Proof)   ▷ Term and Literal Definitions
2: (e_res, unsat_core) ← DRAT-Trim(input, t_lemmas, drat_proof)
3: if e_res = Success then
4:     core_lemmas ← t_lemmas ∩ unsat_core
5:     proof_cert ← Elaborator(core_theory_lemmas, definitions)
6:     val_res ← Validator(proof_cert, core_lemmas, definitions)
7:     if val_res = Success then
8:         return Proof Validation Successful
9:     else
10:        return Theory Lemma Validation Failed
11:    end if
12: else
13:    return DRAT Proof Check Failed
14: end if
```

The key benefit of this approach is that it reduces the trusted code base. If the validator says that a proof certificate is valid, then we can trust that the corresponding lemma is also valid, independent of how the certificate was generated. In other words, we do not need to trust the elaborator, only the validator.

We have implemented elaborators and validators for two SMT-LIB theories: quantifier-free linear real arithmetic (QF_LRA) and quantifier-free uninterpreted functions with equality (QF_UF).

IV. Elaborator and Validator for QF_LRA

Let V be a set of variables. A theory lemma in QF_LRA is of the form $\psi := \bigvee_{i\in[n]}(F_i \bowtie_1 0)$, where each F_i is a linear expression over the variables V and $\bowtie_i \in \{<, \leq, =, >, \geq, \neq\}$. For example, the law of trichotomy $\psi_0 := x > 0 \vee x = 0 \vee x < 0$ is a theory lemma. Validating such a lemma is equivalent to proving that its negation—a conjunction of linear inequalities—is not satisfiable.

Example IV.1. The negation of ψ_0 is $\neg\psi := x \leq 0 \wedge x \neq 0 \wedge x \geq 0$, which can be rewritten as $\neg\psi := (-x \geq 0 \wedge x > 0 \wedge x \geq 0) \vee (-x \geq 0 \wedge -x > 0 \wedge x \geq 0)$, where the inequalities in each disjunct only have either $\geq$ or $>$ as the relational operator.

As shown in Example IV.1, our goal is to create a proof of unsatisfiability for a disjunction of conjunctions of linear inequalities that only involve $\geq$ or $>$ as the relational operators. For a single conjunctive QF_LRA formula, we use Farkas' Lemma to produce the unsatisfiability certificate.

Lemma 1. [16, Farkas' Lemma] A set S of linear inequalities of the form $F_i \{\geq, >\}$ 0 is unsatisfiable if and only if there exists a non-negative linear combination of the inequalities in $S \cup \{1 > 0\}$ deriving either $-1 \geq 0$ or $0 > 0$.

Example IV.2. The formula $\neg\psi$ from Example IV.1 is unsatisfiable if both the disjuncts are unsatisfiable. The expression $1 \cdot (-x \geq 0) + 1 \cdot (x > 0) + 0 \cdot (x \geq 0) \equiv 0 > 0$ is a witness to the unsatisfiability of $-x \geq 0 \wedge x > 0 \wedge x \geq 0$, and $0 \cdot (-x \geq 0) + 1 \cdot (x > 0) + 1 \cdot (x \geq 0) \equiv 0 > 0$ witnesses the unsatisfiability of $-x \geq 0 \wedge -x > 0 \wedge x \geq 0$.

The set of non-negative multipliers for the linear inequalities that derive a trivially false inequality such as $-1 \geq 0$ or $0 > 0$ is called the *Farkas certificate* of unsatisfiability. We reduce the problem of finding the Farkas certificate to solving a linear program. For this purpose, we rely on the following variant of Farkas' Lemma:

Theorem 2. *A conjunction of linear inequalities of the form* $\psi := \bigwedge_{i=1}^{n} F_i \geq 0 \wedge \bigwedge_{j=1}^{m} G_j > 0$ *is unsatisfiable if and only if there exist non-negative constants* $\lambda_1, \ldots, \lambda_n$ *and* $\mu_0, \mu_1, \mu_2, \ldots, \mu_m$ *such that* $\mu_0 + \sum_{i=1}^{n} \lambda_i F_i + \sum_{j=0}^{m} \mu_j G_j \equiv 0$ *with* $\sum_{j=0}^{m} \mu_j = 1$ *(where* $\equiv$ *means that the expressions on both sides are identical).*

Proof. Farkas' Lemma guarantees that ψ is unsatisfiable if and only if one can derive either $-1 \geq 0$ or $0 > 0$ as non-negative linear combination of inequalities in $\psi \cup \{1 > 0\}$.

Let the non-negative linear combination be $D :\equiv \mu_0(1 > 0) + \sum_{i=1}^{n} \lambda_i(F_i \geq 0) + \sum_{j=0}^{m} \mu_j(G_j > 0)$. WLOG, we assume that $D \equiv 0 > 0$ because if $D \equiv -1 \geq 0$, then we set $\mu_0 \leftarrow \mu_0 + 1$ to derive $0 > 0$. Finally, we scale all λ_is and μ_js by a factor of $1/(\sum_{j=0}^{m} \mu_j)$ to ensure that $\sum_{j=0}^{m} \mu_j = 1$. □

The Valido elaborator for QF_LRA produces the Farkas certificate for each core theory lemma by searching for coefficients λ_i and μ_j that satisfy the conditions of Theorem 2. This amounts to solving a system of linear inequalities, which we do using the Simplex algorithm. The generated certificates are stored in a single file for all the core theory lemmas.

Example IV.3. Consider the following *e*DRAT proof fragment.

```
1  (declare-fun x () Real)
2  (define-let aux!0 (* x 1/2))
3  (define-let aux!1 (>= aux!0 0))
4  (define-let aux!2 (< x 0))
5  (define-let aux!3 (> x 0))
6  (define-literal 1 aux!1)
7  (define-literal 2 aux!2)
8  (define-literal 3 aux!3)
9  t 1 2 0
10 t 2 3 0
```

The theory lemma at line 9 is $\psi_6 := x/2 \geq 0 \vee -x > 0$ (which is valid), and that at line 10 is $\psi_7 = x > 0 \vee x < 0$

(which is not valid). On this example input, the elaborator will produce the following output:

```
LINE 9, (0, 1>0), (2, 1), (1, 2)
LINE 10, INVALID LEMMA
```

The first line is the Farkas certificate for ψ_6 (which is at line 9 in the original *e*DRAT proof). The certificate is a list of pairs (`farkas_coefficient`, `literal id`) with an optional term of the form (`farkas_coefficient`, `1>0`) for the Farkas coefficient of $1 > 0$. Thus, the certificate for ψ_6 is $0 \cdot (1 > 0) + 2 \cdot (x/2 \geq 0) + 1 \cdot (-x > 0) \equiv 0 > 0$.

The second line states that the lemma at line 10 of the *e*DRAT proof is invalid.

We have implemented a QF_LRA validator in Lean 4, in around 1300 lines of code. A few important data structures and functions are as follows:

1) A `LinearExpression` is a map `lexpr : Variable → Rat` that maps a variable to its rational coefficient in the expression. A `LinearConstraint` is a pair that consists of a `LinearExpression` and a relational operator, which is either $\geq$ or $>$.
2) A `Model` is a mapping from variables to rationals.
3) Function `evaluate (lexpr : Linear Expression) (m : Model)` computes the value of a `LinearExpression` in a `Model`
4) We define a proposition `isUnsat` as

```
def isUnsat (lemma : List LinearConstraint) (
    m: Model): Prop :=
  match lemma with
  | [] => False
  | (cnstr, lemma') => (evaluate cnstr m) → (
      isUnsat lemma' m)
```

Given a negated lemma $S = \{C_1, \ldots, C_n\}$ as a set of linear constraints, `isUnsat` S is equivalent to

$$\forall \texttt{(m: Model)}, \ \texttt{evaluate}\ C_1\ \texttt{m} \rightarrow \ldots \rightarrow \texttt{evaluate}\ C_n\ \texttt{m} \rightarrow \texttt{False}$$

This proposition says that for every `(m:Model)` at least one of `evaluate` C_i `m` must evaluate to *false*.
5) Given a negated lemma and its Farkas certificate of unsatisfiability, the following function checks whether the certificate is valid.

```
def check_farkas_certificate
    (farkas_coefficients: List Rat)
    (negated_lemma: List LinearConstraint) :
        Bool := ...
```

6) Finally, we proved the following theorem, which shows that function `check_farkas_certificate` is sound:

```
theorem check_farkas_cert_implies_isUnsat (
    check_farkas_certificate
    farkas_coefficients negated_lemma) = true
  → isUnsat negated_lemma := ...
```

The validator first parses the original *e*DRAT proof to collect the definition of each literal and theory lemma. It then parses the certificate file produced by the elaborator and checks every theory Farkas certificate with the function `check_farkas_certificate`. The check is successful if all theory lemmas in the certificate are valid.

V. Elaborator and Validator for QF_UF

QF_UF is one of the simplest theories defined in SMT-LIB. Formulas in QF_UF can include uninterpreted functions, predicates, and equality. A theory lemma in QF_UF is a disjunction of equalities and inequalities between uninterpreted terms. For example, $\psi := x \neq f(y) \vee y \neq g(z) \vee f(x) = f(f(g(z)))$ is a valid theory lemma in QF_UF.

A set of literals F in QF_UF must contain at least one inequality to be inconsistent. The traditional approach to show the inconsistency of F is based on congruence closure, as shown in Algorithm 2. This algorithm builds the smallest congruence relation Eq over the terms of F that includes all input equalities, and then checks whether a negated equality of F is inconsistent with Eq.

Algorithm 2 Congruence Closure Algorithm

```
1: Input: $E$: a finite set of equalities, $D$: a finite set of inequalities
2: Output: Unsat if $E \wedge D$ is not satisfiable, Sat otherwise
3: $T \leftarrow$ All terms occuring in $E \cup D$ (including all the sub-terms)    ▷ Initialization
4: $Eq \leftarrow$ Each $t \in T$ in a singleton class
5: for Each $t = u$ in $E$ with $Eq(t, u) =$ False do    ▷ Process input equalities
6:     $Eq \leftarrow$ Merge classes of $t$ and $u$ in $Eq$
7: end for
8: while $\exists\, C_1, C_2 \in Eq, f(t_1, \ldots, t_n) \in C_1, f(u_1, \ldots, u_n) \in C_2$ such that $C_1 \neq C_2$ and $Eq(t_1, u_1) \wedge \ldots \wedge Eq(t_n, u_n)$ do
9:     $Eq \leftarrow$ Merge classes $C_1$ and $C_2$ in $Eq$
10: end while
11: for each inequality $t \neq u$ in $D$ do    ▷ Check for inconsistency
12:     if $Eq(t, u)$ holds then
13:         return Unsat
14:     end if
15: end for
16: return Sat
```

To check the results of Algorithm 2, it is sufficient to prove that we start with the right initial Eq and that every *Merge Class* operation is sound: that is, when we merge C_1 and C_2 at line 9 of Algorithm 2, the terms in those classes are congruent with respect to the current equivalence relation Eq.

The QF_UF elaborator in Valido generates unsatisfiability certificate based on this idea. Each certificate contains a description of the set of terms T, the initial equalities E, a series of equalities derived from E through congruence, and the inequality from D that led to unsatisfiability. The certificate format is kept simple to simplify parsing. An example is shown in Figure 1.

The certificate consists of the following three parts.

1) Definitions: The certificate starts with the definition of seven terms: four atomic terms including the two Boolean constants `true` and `false` and two uninterpreted constants c_4 and c_0, and three terms built by the application of function f. Each term is identified by its index in this

```
LINE: 16648, CERT
true
false
c_4
c_0
f 3
f 2
f 5
E(3, 5)
E(2, 6)
C(6, 4)
D(2, 4)
```

Fig. 1: Example QF_UF certificate

list. For example, the line `f 3` defines a term of index 4 obtained by applying the uninterpreted function f to the term of index 3. In other words, the term of index 4 is $f(c_0)$.

2) Equalities: After the term definitions, we list equalities from E. Each input equality is written as a line $E(i, j)$ where i and j are two term indices. For example, the line `E(3, 5)` is the equality $c_0 = f(c_4)$. An equality derived by congruence is written similarly but with the letter `C`. In the example `C(6,4)` represents the equality $f(f(c_4)) = f(c_0)$.
3) Inequality: Finally, the last line of the certificate is an inequality, indicated with the letter `D`, between the terms at indices 2 and 4, that is, $c_4 \neq f(c_0)$

The Boolean constants are predefined and included in all certificates (as the first two terms). This enables us to treat uninterpreted predicates as functions from some domain type to the Boolean. For example, a literal of the form $P(x)$ occurring in a theory lemma is treated as $P(x) = true$ in our certificates, and if $\neg P(x)$ occurs, it is converted to $P(x) = false$. This simple trick allows an unmodified congruence closure algorithm to handle uninterpreted predicates (provided we add the built-in inequality $true \neq false$).

The QF_UF validator parses the certificates produced by Valido and checks that they are valid. The central part in the validation process is a union-find data structure implemented in Lean 4 that is used to check that all equalities of the form $C(i, j)$ listed in a certificate are correct, that is, that the two indices i and j denote existing terms and that these two terms are congruent. The validator also checks a similar property for the inequality $D(i, j)$: the two indices i and j must represent existing terms, and the certificate is valid if i and j are in the same equivalence class in the union-find data structure. These checks are implemented in a function `check_certificate`, and the main correctness result follows:

```
def true_certificate (m: Model α β)
        (c: Certificate α β): Prop :=
   m.list_eq_holds c.wft c.base →
      m.diseq_holds c.wft c.conflict

theorem check_certificate_is_sound {α β: Type} [
      BEq β]
   [LawfulBEq β}] (c: Certificate α β})
     (h: Checker.check_certificate c = .ok ()):
       ∀ m, true_certificate m c := by
       ...
```

This states that the function `check_certificate` is sound. If this function succeeds (i.e., it returns `.ok ()`) then the certificate is true in any model `m`. In this theorem, a certificate is parameterized by two types α and β that represent the constants and function symbols in QF_UF terms. The certificate data structure includes a term table, a list of base equalities, a list of derived equalities, and a conflict of the form $D(i, j)$. A model is defined by three components: a domain τ (which is an arbitrary Lean type), a mapping from α to τ that defines the interpretation of constants, and a mapping from β to functions on τ that defines the interpretation of function symbols.

VI. Experiments

We have instrumented cvc5-1.1.1 to produce *e*DRAT proofs. The modifications consist of a new module that prints the *e*DRAT proof and changes to several existing cvc5 modules involved in the creation of input and theory clauses. Most changes were in the CDCL solver employed by cvc5, which is a heavily modified variant of MiniSat.

We have compared the *e*DRAT and Valido toolchain with two other proof formats currently supported by cvc5-1.1.1 on the QF_UF and QF_LRA benchmarks of the SMT-LIB repository [31]. All the experiments were run on a server with 384 GB RAM and 96 cores (48 Intel Xeon Platinum 8259CL CPUs), with a 2.50 GHz CPU frequency. The server runs Amazon Linux 2.

We ran cvc5 with a timeout of 300 seconds with four different proof-generation options: no proofs, proofs in the Alethe-LF (ALF) format, proofs in the LFSC format, and proofs in the *e*DRAT format. Some older versions of cvc5 also support the Alethe format, but this does not appear to be supported anymore in cvc5-1.1.1 and did not work on our benchmarks.

A summary of our experimental results is shown in Table I. The table includes the number of solved problems, the number of proofs successfully produced, and the average runtime on the satisfiable and unsatisfiable problems. A more detailed view of the experimental results is given in Tables II and III.

A. Proof Production Cost

As the table shows, generating proofs in the *e*DRAT format has low overhead. The difference in runtime between baseline cvc5 and cvc5-*e*DRAT on the QF_LRA problems is about 1%. On the QF_UF benchmarks, the average overhead of *e*DRAT proofs is about 16% on satisfiable instances and 27% on unsatisfiable instances. However, the QF_UF benchmark contains many easy problems that are solved in milliseconds (63% of the problems are solved by cvc5 in less than 0.1 s). If we remove these easy problems, the runtime difference between cvc5 and cvc5-*e*DRAT is less than 10%. In total, cvc5 and cvc5-*e*DRAT solve the same number of problems in all categories, apart from the class of satisfiable QF_LRA

TABLE I: Summary of Experiments

	QF_LRA						QF_UF					
	Solved Problems				Avg. Runtime (s)		Solved Problems				Avg. Runtime (s)	
Proof Mode	Unsat	Proofs	Sat	Unsolved	Sat	Unsat	Unsat	Proofs	Sat	Unsolved	Sat	Unsat
None	639		902	212	22.046	31.890	4353		3142	8	0.245	0.723
*e*DRAT	639	639	899	215	22.267	31.481	4353	4353	3142	8	0.286	0.924
ALF	621	591	876	256	32.185	56.267	4345	4335	3142	16	0.361	5.543
LFSC	623	503	876	254	32.412	85.825	4344	4283	3142	17	0.353	12.786

TABLE II: Experiment results on QF_LRA benchmark. All sizes are in MBs and times are in seconds. The averages are taken over the benchmark where the corresponding proof was successfully checked. The column ✓ represents the number of proofs that were successfully checked.

		cvc5 + Proof Generation Time				LFSC Proof			ALF Proof			*e*DRAT Proof		
Family	#	No Proof	LFSC	ALF	EDRAT	✓	Size	Time	✓	Size	Time	✓	Size	Time
Heizmann	29	87.725	196.195	159.793	85.504	15	61.251	55.099	19	40.978	105.052	29	14.491	10.066
LassoRanker	91	77.518	178.089	126.096	77.762	61	72.289	29.124	87	20.665	68.470	91	10.216	4.437
sc	35	44.889	194.576	81.503	42.856	20	225.166	104.933	32	23.178	129.505	35	5.735	1.339
uart	34	32.989	224.543	126.002	31.094	11	206.111	87.419	26	44.383	558.786	34	6.753	3.508
clock	36	20.454	81.828	28.276	19.925	29	70.890	20.705	36	6.158	25.426	36	0.938	1.027
latendresse	1	18.494	36.977	29.516	17.963	0	NA	NA	1	3.236	1.702	1	0.460	28.339
miplib	11	14.988	111.516	59.597	14.979	7	72.550	31.241	10	25.839	175.199	11	10.351	78.223
tta_startup	45	8.353	60.505	33.536	8.206	34	29.170	15.290	43	12.041	158.288	45	3.314	0.990
blending	9	3.886	300.212	114.993	4.301	0	NA	NA	9	129.184	119.071	9	15.932	49.526
TM	1	0.525	2.305	1.827	0.923	0	NA	NA	1	1.147	1.665	1	0.680	0.274
sal	96	0.094	1.731	0.655	0.120	96	2.222	1.354	96	0.680	0.941	96	0.162	0.178
spider	42	0.061	1.351	0.357	0.085	42	2.876	0.818	42	0.295	0.110	42	0.082	0.148
robotics	12	0.011	0.037	0.041	0.010	12	0.065	0.047	12	0.048	0.013	12	0.000	0.127
check	1	0.009	0.108	0.055	0.011	1	0.144	0.038	1	0.109	0.045	1	0.009	0.132
meti-tarski	150	0.007	0.013	0.011	0.007	150	0.008	0.011	150	0.007	0.011	150	0.001	0.128
keymaera	21	0.006	0.010	0.009	0.006	21	0.003	0.010	21	0.003	0.010	21	0.000	0.128

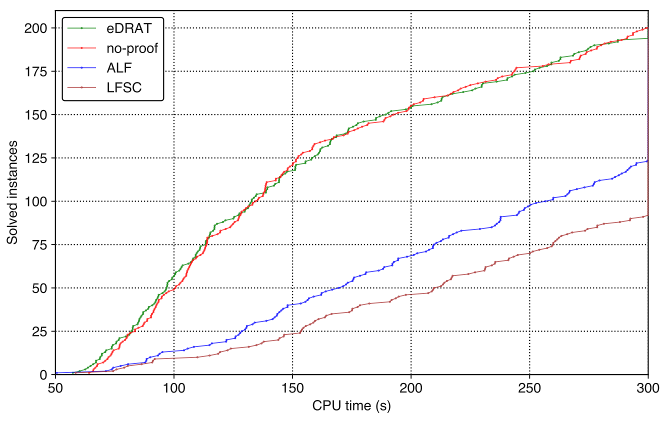
TABLE III: Experiment results on QF_UF benchmark. All sizes are in MBs and times are in seconds. The averages are taken over the benchmark where the corresponding proof was successfully checked. The column ✓ represents the number proofs that were successfully checked.

		cvc5 + Proof Generation Time				LFSC Proof			ALF Proof			*e*DRAT Proof		
Family	#	No Proof	LFSC	ALF	EDRAT	✓	Size	Time	✓	Size	Time	✓	Size	Time
Rodin	20	0.006	0.008	0.007	0.006	20	0.002	0.010	20	0.002	0.010	20	$\leq$ 0.001	0.079
Goel	229	0.209	9.406	8.549	0.232	217	0.413	1.660	226	0.311	0.220	229	0.092	0.091
CLEARSY	11	0.013	0.135	0.084	0.015	11	0.101	0.064	11	0.086	0.028	11	0.008	0.078
eq_diamond	100	0.022	0.342	0.340	0.043	100	0.051	0.067	100	0.044	0.031	100	0.055	0.086
NEQ	45	3.589	141.322	27.222	3.934	24	83.305	26.559	45	39.562	139.976	45	4.217	1.022
PEQ	38	7.394	171.559	61.164	8.418	20	136.427	46.852	34	51.902	129.771	38	20.327	3.919
SEQ	39	1.150	85.259	12.251	1.392	34	135.283	72.992	39	20.125	65.002	39	3.398	0.729
QG-class	3859	0.315	8.956	3.993	0.390	3854	10.486	23.253	3857	6.349	31.858	3859	0.495	0.224
TypeSafe	3	0.006	0.009	0.008	0.006	3	0.001	0.011	3	0.002	0.010	3	$\leq$ 0.001	0.078

problems. In this class, four problems are solved by cvc5 but not by cvc5-*e*DRAT and one problem is solved by cvc5-*e*DRAT but not by cvc5. This difference is most likely due to random variation caused by the operating system as all four take a runtime close to the timeout, rather than caused by the *e*DRAT proof generation.

The LFSC and ALF formats are more expensive, and the overhead depends on the theory. On QF_LRA, cvc5 fails to solve about 40 problems when using either format. On QF_UF producing either LFSC or ALF proofs doubles the number of timeouts. The runtime overhead is around 45% for both LFSC and ALF on satisfiable problems (on both QF_LRA and QF_UF). For the unsatisfiable problems, the overhead varies depending on proof-format and theory: on QF_LRA, LFSC incurs an overhead of 2.7x, and ALF is close to 2x slower than baseline cvc5. On QF_UF, the overhead is 7x for ALF and 17x for LFSC. The larger overhead on QF_UF is due to the fact that LFSC and ALF are not compatible with a symmetry-breaking procedure that baseline cvc5 employs [15]. Symmetry breaking is effective on the QF_UF benchmarks, but

(a) QF_LRA



(b) QF_UF

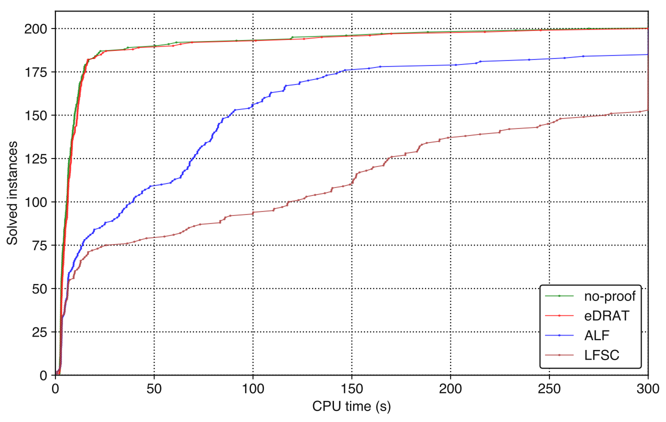


Fig. 2: Runtime on hardest problems

it must be disabled when CVC5 produces LFSC or ALF proofs. The *e*DRAT format is compatible with symmetry breaking and does not suffer from this disadvantage. We also see that both CVC5-LFSC and CVC5-ALF can solve a problem (i.e., print "unsat") but fail to generate a proof within the timeout. This happens because LFSC and ALF proofs are generated after CVC5 finds a problem to be unsat. After the problem is solved, CVC5 performs backward dependency analysis to construct a proof and export it to the LFSC or ALF format [4]. Both backward analysis and conversion to the external format can be expensive and cause a timeout.

Both the QF_UF and QF_LRA benchmarks contain a large number of easy problems that are solved in milliseconds. Figure 2 compares the runtime of our four CVC5 variants on the 200 problems that take the longest for baseline CVC5 to solve. The plots show that CVC5 and CVC5-DRAT have similar performance. CVC5-ALF and CVC5-LFSC are slower and timeout on several problems, but CVC5-ALF is more efficient than CVC5-LFSC.

B. Proof Size and Proof Checking Time

Figure 3 compares the proof sizes for different problem families in the QF_LRA and QF_UF benchmarks. The differences between the three formats vary with the theory and the problem family. Overall, *e*DRAT is more compact, except for a few problems. In QF_UF, ALF proofs are 2x larger than *e*DRAT proofs, and LFSC proofs are 11x larger than ALF proofs on average. In QF_LRA, ALF proofs are 4x larger than *e*DRAT proofs, and LFSC proofs are 11x larger than ALF proofs. Some of the size difference is due to the fact that ALF and LFSC include preprocessing steps, but this is significant mostly on easy problems. On hard problems, preprocessing represents a small part of the solver work, and proof steps related to resolution and theory lemmas dominate.

We validated the proofs with the appropriate checker. For LFSC, we used LFSCC[2]; for ALF, we use `alfc`[3]; and for *e*DRAT, we used Valido. All proofs were valid. Figure 4 shows the average proof checking time per benchmark family. For *e*DRAT, the graphs include the runtime of Valido (in Rust) and the certificate checkers (in Lean). On a few QF_LRA proofs, Valido is slower than the ALF checker (e.g., in the Latendresse family). This happens when theory lemmas are large (several hundreds of atoms per lemma) and our Simplex implementation is slow at computing Farkas certificates. Most proofs do not have such large lemmas. The ALF and LFSC checkers are also faster than Valido in some families of QF_UF problems, but the proofs in these families are small, and all checkers validate them in less than 0.1 s. On such small proofs, the cost of a call to `DRAT-trim` is a limiting factor for Valido. But overall, *e*DRAT proof checking is 3x and 15x faster than LFSC and ALF proof checking, respectively, in QF_LRA benchmarks, and 80x and 120x faster than LFSC and ALF, respectively, in QF_UF benchmarks. As one would expect, checking unsatisfiability certificates is cheaper than constructing unsat cores and certificates in the first place. The runtime of the certificate checker in Lean is smaller than the cost of the elaborator and `DRAT-trim` in all problem families. We also note that only a small fraction of all the theory lemmas included in the proof are part of the unsat core. Figure 5 shows the number of theory lemmas in the core compared with the total number of theory lemmas in the *e*DRAT file. Only lemmas in the core must be checked by Valido. On average, 1/8 of the QF_LRA theory lemmas and 1/2 of the QF_UF theory lemmas are in the core.

VII. Related and Future Work

Our results show that the DRAT proof format can be extended from SAT to SMT while preserving its efficiency. Compared with other proof formats currently supported by CVC5, *e*DRAT is the cheapest to generate. Although the *e*DRAT proofs are not detailed, it is still possible to efficiently check them by combining unsat core construction and specialized checkers for theory lemmas.

Otoni, et al. [27] present a proof system for OpenSMT that also combines DRAT with theory-specific checkers. A difference with our approach is that OpenSMT is modified to produce unsatisfiability certificates for each theory lemma,

[2]https://github.com/cvc5/LFSC
[3]https://github.com/cvc5/alfc

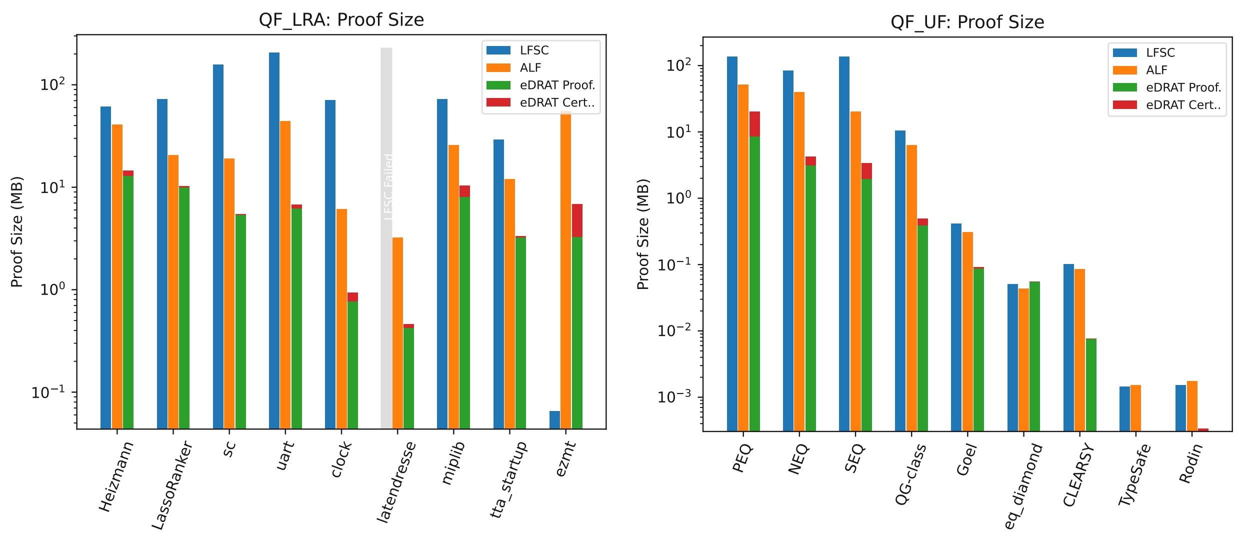


Fig. 3: Proof Sizes

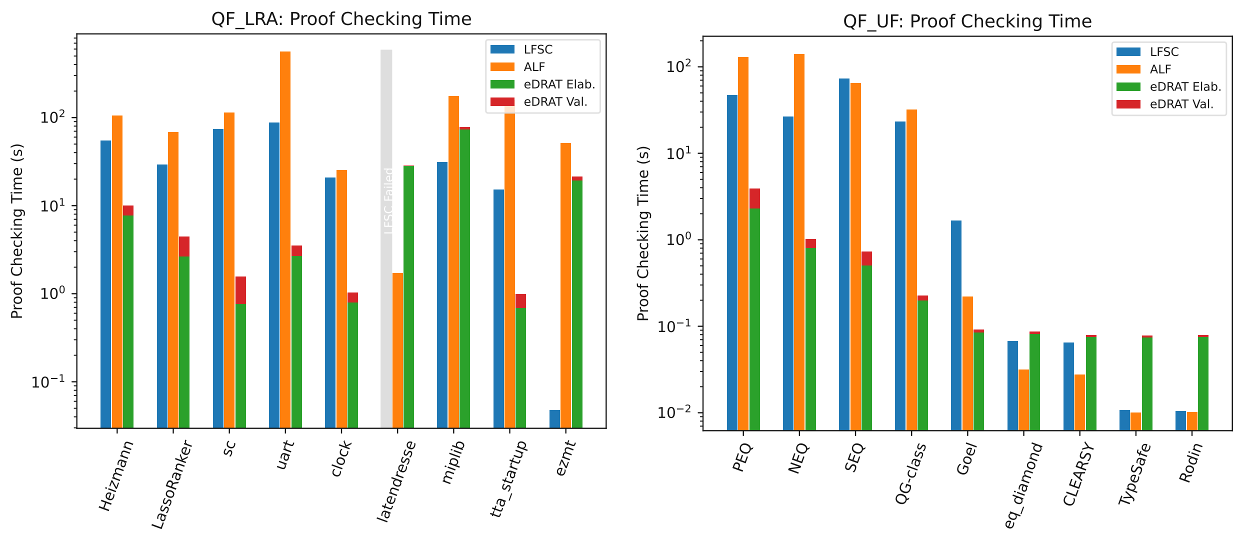


Fig. 4: Proof Checking Time

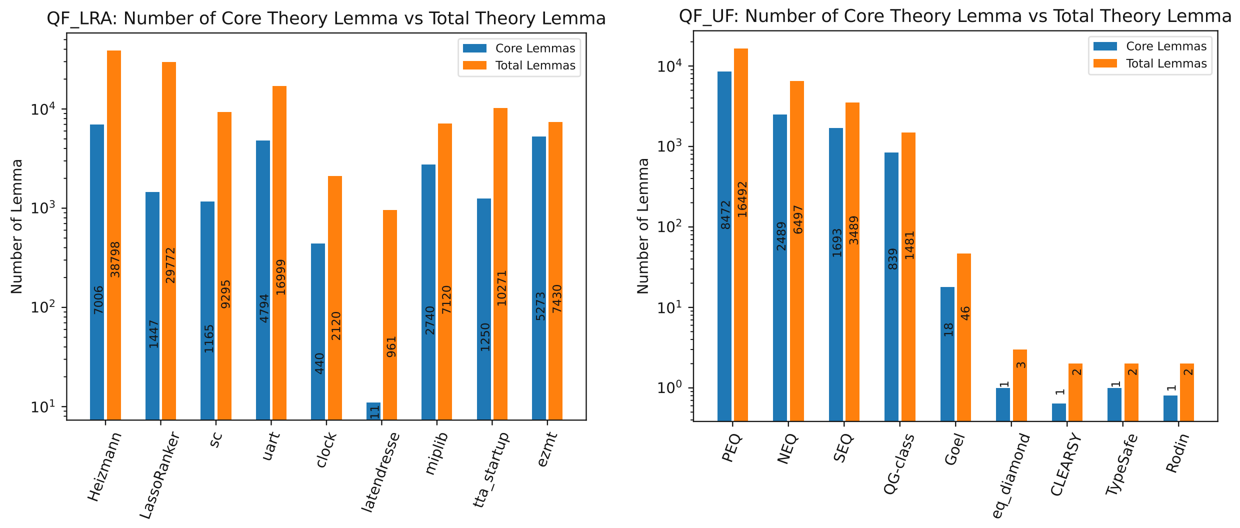


Fig. 5: Core Theory Lemmas

whereas we use an external elaborator to construct these certificates. Because we do not modify the CVC5 theory reasoning engines, we can efficiently produce *e*DRAT proofs for any theory supported by CVC5 (even though we cannot yet validate all of them). Another difference is that Otoni, et al. can the check conversion from SMT to CNF using a two-phase algorithm. The first phase checks a conversion from SMT to a DAG format (not defined in the paper), and the second phase checks Tseitin-style CNF conversion. This is more than what we can do with eDRAT, but it does not seem to be sufficient for the rewriting steps employed by CVC5. It is not clear from [27] how the simplifications that cvc5 heavily uses (such as the elimination of if-then-else, variable elimination, normalization of terms, and many other rewriting steps) could be handled. Finally, [27, Table II] shows that the overhead of their proof-production method is significant (e.g., 25% fewer solved instances in QF_LRA), while the main goal of eDRAT is to make proof generation as cheap as possible.

Another DRAT extension to SMT is presented by Feng, et al. [17]. This approach is specialized for satisfiability modulo monotonic theories. In this setting, predicates are monotonic relations over Boolean variables, and Feng, et al. use this property to build propositional DRAT proofs of theory lemmas. Like VALIDO, these extensions of DRAT for SMT offer proofs at low cost. The numbers reported in Otoni, et al. and Feng, et al. show that their proof generation techniques are efficient.

Currently, the main limitation of our approach is that it starts from a CNF formula. *e*DRAT is not adequate for representing proofs of preprocessing and conversion of formulas to clauses. We are considering three options to bridge this gap:

- Modify CVC5 to produce proofs of only its preprocessing steps in, say, the ALF format. This is probably the easiest approach but it has limitations. For example, as discussed in Sec. VI, some useful preprocessing steps must be disabled, and scalability remains to be evaluated.
- Use translation validation [28]. One can see preprocessing and conversion to CNF as a compilation process. Correctness amounts to showing that this compilation preserves satisfiability, and translation validation can be adapted to this problem. An issue is that this may require the solver to produce hints to enable this approach.
- Implement a provably correct preprocessor, say, in Lean. This may require the most effort, but it could provide the most benefit. One issue with this option is the cost of maintaining and updating the preprocessor as new theories and possibly new simplification techniques are discovered.

VIII. Conclusion

*e*DRAT extends the well-known DRAT format of SAT to SMT. Our experiments show that *e*DRAT proofs can be produced efficiently and can be efficiently validated, which makes routine use of proof-producing SMT solvers more practical. In future work, we will extend the VALIDO tool chain to cover more theories, and we will extend the approach to include proofs of preprocessing.

Acknowledgment

S Hitarth was partially supported by the Madrid Regional Government (César Nombela grant 2023-T1/COM-29001), MCIN/AEI/10.13039/501100011033/FEDER, and EU (grant PID2022-138072OB-I00).

References

[1] B. Andreotti, H. Lachnitt, and H. Barbosa. Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13993, pages 367–386. Springer Nature Switzerland, 2023.

[2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018.

[3] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. CVC5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *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*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[4] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, and C. Barrett. Flexible proof production in an industrial-strength SMT solver. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning*, pages 15–35, Cham, 2022. Springer International Publishing.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at https://smt-lib.org.

[6] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Chapter 33. Satisfiability Modulo Theories. In A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2021.

[7] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT solver. In R. A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 151–156, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing.

[9] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 model checker. In *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing.

[10] J. Christ, J. Hoenicke, and A. Nutz. SMTInterpol: An interpolating SMT solver. In A. Donaldson and D. Parker, editors, *Model Checking Software*, pages 248–254, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] L. de Moura and N. Bjørner. Proofs and refutations, and z3. In *The LPAR 2008 Workshops: KEAPPA and IWIL 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, November 2008.

[12] L. de Moura and N. Bjørner. 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*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: An Appetizer. In M. V. M. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902, pages 23–36. Springer Berlin Heidelberg, 2009.

[14] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In A. Platzer and G. Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[15] D. Déharbe, P. Fontaine, S. Merz, and B. Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 222–236, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] J. Farkas. Theory of simple inequalities. *Journal for pure and applied mathematics (Crelles Journal)*, 1902(124):1–27, 1902.

[17] N. Feng, A. J. Hu, S. Bayless, S. M. Iqbal, P. Trentin, M. Whalen, L. Pike, and J. Backes. Drat proofs of unsatisfiability for sat modulo monotonic theories. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–23, Cham, 2024. Springer Nature Switzerland.

[18] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[20] A. Goel and K. Sakallah. AVR: Abstractly verifying reachability. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–422, Cham, 2020. Springer International Publishing.

[21] M. J. H. Heule. The DRAT format and DRAT-trim checker.

[22] J. Hoenicke and T. Schindler. A simple proof format for SMT. In D. Déharbe and A. E. J. Hyvärinen, editors, *Satisfiability Modulo Theories, 2022*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, August 2022.

[23] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 547–553, Cham, 2016. Springer International Publishing.

[24] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[25] J. Marsuqes-Sliva. Chapter 4. Conflict-Driven Clause Learning. In A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2021.

[26] K. L. McMillan and O. Padon. Ivy: A multi-modal verification tool for distributed algorithms. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification*, pages 190–202, Cham, 2020. Springer International Publishing.

[27] R. Otoni, M. Blicha, P. Eugster, A. E. J. Hyvärinen, and N. Sharygina. Theory-specific proof steps witnessing correctness of SMT executions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 541–546, 2021.

[28] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[29] N. Rungta. A billion SMT queries a day (invited paper). In S. Shoham and Y. Vizel, editors, *Computer Aided Verification*, pages 3–18, Cham, 2022. Springer International Publishing.

[30] H. Schurr, M. Fleury, H. Barbosa, and P. Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In C. Keller and M. Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

[31] SMT-LIB. The Satisfiability Modulo Theories Library. https://smtlib.cs.uiowa.edu/. Accessed on March 15, 2024.

[32] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[33] A. Stump, A. Reynolds, C. Tinelli, A. Laugesen, H. E. III, C. Oliver, and R. Zhang. LFSC for SMT proofs: Work in progress. In D. Pichardie and T. Weber, editors, *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving, PxTP 2012, Manchester, UK, June 30, 2012*, volume 878 of *CEUR Workshop Proceedings*, pages 21–27. CEUR-WS.org, 2012.

[34] N. Wetzler, M. J. H. Heule, and W. A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014*, volume 8561, pages 422–429. Springer International Publishing, 2014.

Solving String Constraints with Concatenation Using SAT

Kevin Lotz*, Amit Goel†, Bruno Dutertre‡, Benjamin Kiesl-Reiter§, Soonho Kong‡, and Dirk Nowotka*
*Department of Computer Science, Kiel University, Kiel, Germany, {kel, dn}@informatik.uni-kiel.de
†Amazon Web Services, Portland, OR, USA, amgoel@amazon.com
‡Amazon Web Services, Santa Clara, CA, USA, {dutebrun, soonho}@amazon.com
§Amazon Web Services, Munich, Germany, benkiesl@amazon.com

***Abstract*—We present a decision procedure for solving quantifier-free first-order formulas over the theory of strings, involving equality, regular constraints, and concatenation of string terms. Our approach uses an eager reduction to the Boolean satisfiability problem and extends the NFA2SAT string solver. We describe a novel SAT encoding for word equations that iteratively expands the search space and leverages incremental SAT solving. For unsatisfiable formulas, we estimate the bounds on the smallest solution from arithmetic constraints derived from word equations. An experimental evaluation shows that our approach is competitive with state-of-the-art string solvers and complements existing methods in string solving.**

I. Introduction

Reasoning on string manipulation is a crucial aspect of ensuring software correctness. In recent years, a variety of tools, known as *string solvers*, have been developed to automate decision procedures for various logical theories over strings. Advancements in string solving have been driven by web-application security [16], [27], [29] and model checking [11]. These fields rely on automated reasoning on strings to identify critical security vulnerabilities. More recently, string solving has been used to verify security properties of cloud access policies [1], [26].

The theory of strings draws upon combinatorics on words [25], [12], [4], [22]. Central to this theory are word equations, which are expressions that equate two strings constructed by *concatenating* variables and constant words. Solving word equations amounts to finding substitutions for the variables that make the two sides of the equation identical. For example, we find a solution for $a \cdot \mathsf{x} \doteq \mathsf{y} \cdot a$ by substituting both variables x and y with a. Solving word equations is decidable [22], [8], [28], but the decision procedures resulting from the theoretical results are too expensive to be practical. To overcome this limitation, string solvers employ heuristic approaches and impose restrictions on the constraint languages to achieve scalability in practical use cases.

Most modern string solvers [23], [2], [14], [5], [6], [21] are built upon the *CDCL(T)* paradigm, also called *lazy* solving. This framework operates in two steps: first, a SAT solver searches for a model of the propositional structure of a formula, and second, a theory solver decides whether this model is consistent in a background theory T. An alternative approach is *eager* solving, which encodes the input problem into a single propositional formula. In the context of string solving, eager approaches were first explored by the WOORPJE [7] solver for word equations and its extension to regular constraints [18].

In previous work [20], we presented the eager string solver NFA2SAT that decides the satisfiability of formulas within a restricted logical fragment, which includes regular constraints and equality between strings but excludes concatenation of string terms. The solver is complete on this fragment, but it supports a less expressive logic compared to other string solvers. Here, we bridge this gap by extending NFA2SAT's decision procedure to support word equations.

The NFA2SAT procedure sets bounds on the lengths of all string variables occurring in a formula, encodes the bounded problem into a propositional formula, and tests its satisfiability. If the formula is unsatisfiable, the procedure iterates by incrementally increasing these bounds until either a solution is found, or the bounds exceed the theoretical length of the minimal solution to the formula, at which point the formula is declared unsatisfiable.

To enable support for word equations, we introduce a new method to encode the satisfiability problem of bounded word equations into propositional logic. We also prove an alphabet-reduction result, which we use to obtain a small alphabet that is sufficiently large to preserve satisfiability. This reduction is critical to ensure the practicality of the encoding because it reduces the size of the propositional formula, thereby allowing for more efficient SAT solving. We then propose an incomplete but practical approach to detecting unsatisfiability, by analyzing linear integer equations over the lengths of the string variables that occur in word equations. An experimental evaluation on a large set of benchmarks shows that our approach is competitive with state-of-the-art string solvers and works well as a complement to lazy solvers.

II. Preliminaries

A *word* is a finite sequence $w = w_1 \cdots w_n$ where each w_i is a symbol in a finite alphabet A. We denote by $|w| = n$ the length of word w. The set of all words over A is denoted by A^*. We denote by $w \cdot w'$ the concatenation of two words w and w', and we denote the empty word by ε. A word u is called a *factor* of w if w can be written $v \cdot u \cdot v'$. It is called a *prefix* (*suffix*) if $v = \varepsilon$ ($v' = \varepsilon$). We use $|w|_a$ to denote the number of occurrences of symbol a in word w. We fix

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_9

an alphabet $\Sigma = \{ a, b, c, \dots \}$ of *constants* and an alphabet $\Gamma = \{ \mathsf{x}, \mathsf{y}, \mathsf{z}, \dots \}$ of *variables*. A word $w \in \Sigma^*$ is called a *constant word* and a word $\alpha \in (\Sigma \cup \Gamma)^*$ is called a *pattern*.

A *word equation* $\alpha \doteq \beta$ is a pair of patterns α, β. A *regular constraint* $\alpha \mathrel{\dot\in} R$ consists of a pattern $\alpha \mathrel{\dot\in} (\Sigma \cup \Gamma)^*$ and a regular expression R over the alphabet Σ. Let $h : (\Sigma \cup \Gamma)^* \to \Sigma^*$ be a morphism that is constant for all $c \in \Sigma$, i.e., $h(c) = c$. Then h is a solution of $\alpha \doteq \beta$ (written $h \models \alpha \doteq \beta$) if $h(\alpha) = h(\beta)$, and a solution of $\alpha \mathrel{\dot{\neq}} \beta$ (written $h \models \alpha \mathrel{\dot{\neq}} \beta$) if $h(\alpha) \neq h(\beta)$. Similarly, h is a solution of $\alpha \mathrel{\dot\in} R$ (written $h \models \alpha \mathrel{\dot\in} R$) if $h(\alpha) \in \mathcal{L}(R)$, and a solution of $\alpha \mathrel{\dot\notin} R$ (written $h \models \alpha \mathrel{\dot\notin} R$) if $h(\alpha) \notin \mathcal{L}(R)$, where $\mathcal{L}(R)$ denotes the regular language defined by R. Any substitution of the variables $h : \Gamma \to \Sigma^*$ can be canonically extended to a morphism, and vice versa. We therefore use the terms substitution and morphism interchangeably.

A function $l : \Gamma \to \mathbb{N}$ that assigns a length $l(\mathsf{x})$ to each variable $\mathsf{x} \in \Gamma$ is called a *length assignment*. Given a length assignment l, we use $\overrightarrow{\mathsf{x}}^l$ to refer to the sequence $\mathsf{x}[1] \cdots \mathsf{x}[l(\mathsf{x})]$ over the alphabet $\overrightarrow{\Gamma} = \{ \mathsf{x}[k] \mid \mathsf{x} \in \Gamma, k \in \mathbb{N} \}$. In this sequence, the $\mathsf{x}[i]$ can be interpreted as variables ranging over Σ, that is, each $\mathsf{x}[i]$ denotes a single character of Σ. We lift this definition to patterns with $\overrightarrow{w\alpha}^l = w \cdot \overrightarrow{\alpha}^l$ and $\overrightarrow{\mathsf{x} \cdot \alpha}^l = \overrightarrow{\mathsf{x}}^l \cdot \overrightarrow{\alpha}^l$ for all $w \in \Sigma^*$. For a word equation $\alpha \doteq \beta$, a length assignment l with $|\overrightarrow{\alpha}^l| = n = |\overrightarrow{\beta}^l|$ induces an equivalence relation of the positions $1, \dots, n$. Two positions i, j are *equivalent* under l, written as $i \sim_l j$, if $\overrightarrow{\alpha}[i] = \overrightarrow{\alpha}[j]$, $\overrightarrow{\beta}[i] = \overrightarrow{\beta}[j]$, or $\overrightarrow{\alpha}[i] = \overrightarrow{\beta}[j]$. If $i \sim_l j$, then characters at position i and j must be equal in any solution that is consistent with the length assignment. We call h an l-substitution if $|h(\mathsf{x})| = l(\mathsf{x})$ for all $\mathsf{x} \in \Gamma$. If h is an l-substitution, then h is a solution if and only if $h(\alpha)[i] = h(\alpha)[j]$ for all i, j with $i \sim_l j$. In that case, we call h an *l-solution*. Solving word equations by assigning a constant from Σ to every $\mathsf{x}[1] \cdots \mathsf{x}[l(\mathsf{x})]$ for all $\mathsf{x} \in \Gamma$ to find a morphism that satisfies the above condition is also known as *filling the positions* [15], [25].

We consider quantifier-free first-order formulas in which all atoms are word equations or regular constraints, or can be reduced to them. For such a formula ψ, we use $\mathrm{atoms}(\psi)$ to denote the set of atoms that occur in ψ, $\mathrm{vars}(\psi)$ to denote the set of variables in ψ, and $\Sigma(\psi)$ to denote the set of constants occurring in ψ. A substitution $h : \Gamma \to \Sigma^*$ is called a *model* of ψ, written $h \models \psi$, if ψ evaluates to true under h using the standard semantics of Boolean connectives. We assume throughout the paper that ψ is in negative normal form (NNF), that is, negations occur only in front of atoms. The literals of ψ can be of the form $\alpha \doteq \beta$, $\neg(\alpha \doteq \beta)$, $\alpha \mathrel{\dot\in} R$, or $\neg(\alpha \mathrel{\dot\in} R)$. We use $\alpha \mathrel{\dot{\neq}} \beta$ and $\alpha \mathrel{\dot\notin} R$ as short-hand notation for $\neg(\alpha \doteq \beta)$ and $\neg(\alpha \mathrel{\dot\in} R)$, respectively. We call ψ *conjunctive* if it is a conjunction of literals. We say ψ is in *normal form* if all literals have the form $\alpha \doteq \beta$, $\mathsf{x} \in R$, $\mathsf{x} \notin R$, or $\mathsf{x} \mathrel{\dot{\neq}} \mathsf{y}$. For every formula ψ there exists an equisatisfiable formula ψ' in normal form. We construct it by rewriting literals of the form $\alpha \mathrel{\dot{\neq}} \beta$ to $\alpha = \mathsf{t}_\alpha \wedge \beta = \mathsf{t}_\beta \wedge \mathsf{t}_\alpha \mathrel{\dot{\neq}} \mathsf{t}_\beta$, and literals of the form $\alpha \mathrel{\dot\in} R$ ($\alpha \mathrel{\dot\notin} R$) to $\alpha = \mathsf{t}_\alpha \wedge \mathsf{t}_\alpha \mathrel{\dot\in} R$ ($\alpha = \mathsf{t}_\alpha \wedge \mathsf{t}_\alpha \mathrel{\dot\notin} R$), where t_α and t_β are fresh variables. We have $\Sigma(\psi) = \Sigma(\psi')$ and $\mathrm{vars}(\psi) \subseteq \mathrm{vars}(\psi')$. If $h \models \psi'$, then $h \models \psi$, and if $h \models \psi$, then $h' \models \psi'$, where h' extends h by setting $h'(\mathsf{t}_\alpha) = h(\alpha)$.

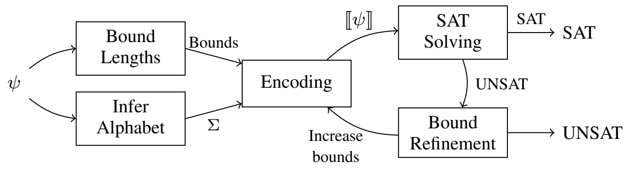


Fig. 1: Overview of the decision procedure.

$$\begin{aligned}
F &:= F \vee F \mid F \wedge F \mid \neg F \mid \text{Atom}\\
\text{Atom} &:= t_{\mathsf{str}} \mathrel{\dot\in} RE \mid \mathsf{t}_{\mathsf{str}} \doteq \mathsf{t}_{\mathsf{str}}\\
RE &:= RE \cup RE \mid RE \cdot RE \mid RE^* \mid RE \cap RE \mid ? \mid w\\
t_{\mathsf{str}} &:= \mathsf{x} \mid w \mid t_{\mathsf{str}} \cdot t_{\mathsf{str}}
\end{aligned}$$

Fig. 2: Syntax: x denotes a variables, w denotes a word of Σ^*, and RE denotes a regular expression, where ? is the wildcard character.

III. The Decision Procedure

Our decision procedure accepts formulas in the syntax given in Figure 2. The procedure is based on an eager reduction to the Boolean satisfiability problem. This reduction assumes fixed upper bounds on the lengths of variables and translates the input problem into a propositional formula that is equi-satisfiable for these bounds. It builds upon the string solver NFA2SAT, which we will review first.

The NFA2SAT procedure is depicted in Figure 1. It begins by assigning an initially small upper bound $b(\mathsf{x})$ on the length of all string variables x. Subsequently, the solver constrains the search space for substitutions to a small alphabet Σ that preserves satisfiability. If the problem is satisfiable in any superset of Σ, it remains satisfiable in Σ. Therefore, the procedure only needs to consider substitutions that map variables to words in the reduced alphabet Σ^*.

For the given bounds and alphabet, NFA2SAT encodes the first-order formula ψ into a propositional formula $[\![\psi]\!]$. If $[\![\psi]\!]$ is satisfiable, NFA2SAT declares ψ satisfiable. If $[\![\psi]\!]$ is unsatisfiable, then there are no solutions that satisfy ψ within the given upper bounds. In this case, NFA2SAT increases the bounds on the variable lengths and incrementally encodes the problem for these new bounds. This incremental encoding produces new clauses without discarding those from the previous SAT solver invocation, leveraging the benefits of incremental SAT solving under assumptions [10].

This process repeats until either the bounds are sufficiently large for $[\![\psi]\!]$ to be satisfiable, or unsatisfiability can be concluded based on the *small model property* stating that if ψ is satisfiable, then there exists a smallest model. We can compute bounds on the variable lengths in the smallest model and compare them to the bounds that resulted in the unsatisfiable encoding. If the bounds of the unsatisfiable encoding exceed those of the smallest model, then no solution exists.

To improve efficiency, NFA2SAT utilizes the unsat core from the last SAT solver call to refine variable bounds and handle unsatisfiability. It first computes the powerset of the (typically small) set of literals encoded in the unsat core. For each subset in the powerset, the solver calculates the small model bounds of the conjunction of the literals in that subset. Finally, take the maximum among all these computed bounds. The iteration over the powerset is necessary because the core is not necessarily minimal. If the bounds used for the last call to the SAT solver exceed the maximum, then increasing the bounds will not eliminate the unsat core and we can then conclude that ψ is not satisfiable. Otherwise, only the bounds of the variables occurring in a literal encoded in the unsat core are increased.

We extend the existing NFA2SAT procedure to support arbitrary concatenation, which amounts to solving word equations after conversion to normal form. Our new procedure includes three new components:

1) **Alphabet Reduction:** NFA2SAT narrows the search space by determining a small alphabet in which the formula is satisfiable if it is in any larger alphabet. In Section IV we show that adding one character to $\Sigma(\psi)$ for each $\not\doteq$ atom is sufficient to preserve satisfiability.
2) **Encoding of Word Equations:** In Section V, we introduce a new encoding that translates word equations into propositional logic in a way that is compatible with NFA2SAT's incremental framework.
3) **Handling Unsatisfiability:** For cases where no solution exists, we propose a simple but incomplete technique to compute the bounds of the small model if the unsat core contains (encoded) word equations. This approach is detailed in Section VI.

IV. Alphabet Reduction

A model for a string formula is a mapping from variables to constant words in an alphabet Σ, which can depend on the context. For example, in the SMT-LIB standard [3], Σ is a subset of the Unicode alphabet that contains $196,607$ symbols. In most cases, a model for satisfiable formulas can be constructed using only a small subset of Σ. Restricting the search space to such a subset is essential for our propositional encoding to be practical. Let $\Sigma(\psi)$ be the set of all constants occurring in ψ. If ψ does not contain string concatenation, it is satisfiable if and only if it is satisfiable in the alphabet $\Sigma(\psi)$ augmented with one additional character per variable [20]. When string concatenation is allowed, we instead need one additional character per negated equation. We show this result by fixing a solution $h : \Gamma \rightarrow \Sigma^*$ over any alphabet Σ and constructing a new solution $h' : \Gamma \rightarrow (\Sigma(\psi) \cup A)^*$, where A is disjoint from $\Sigma(\psi)$ and has a cardinality equal to the number of inequalities in ψ. The construction is based on the method of filling the positions.

Given a word equation $\alpha \doteq \beta$ and a length assignment l, we lift the equivalence relation $\sim_l$ to the elements of $\overrightarrow{\Gamma} \cup \Sigma$. Two elements $r, s \in \Sigma \cup \overrightarrow{\Gamma}$ are equivalent (under l), written as $r \sim_l s$, if there are i, j with $i \sim_l j$ and $\overrightarrow{\alpha}^l[i] = r \wedge \overrightarrow{\alpha}^l[j] = s$ or $\overrightarrow{\beta}^l[i] = r \wedge \overrightarrow{\beta}^l[j] = s$, or $\overrightarrow{\alpha}^l[i] = r \wedge \overrightarrow{\beta}^l[j] = s$. This defines an equivalence relation on $\Sigma \cup \overrightarrow{\Gamma}$. We use $[r]_{\sim_l}$ to refer to the transitive reflexive closure of $r \in \Sigma \cup \overrightarrow{\Gamma}$ under l. If l is clear from the context, we simply write $r \sim s$ and $[r]_\sim$. An l-solution h maps every class $[r]_\sim$ to exactly one constant c in the sense that $h(\mathsf{x})[j] = c$ for all $\mathsf{x}[j] \in [r]_\sim$. If $[r]_\sim$ contains a constant character then c is that character. For example, let $\mathsf{x} \cdot a \doteq a \cdot \mathsf{y}$ be a word equation. The length assignment $l(\mathsf{x}) = 3$ and $l(\mathsf{y}) = 3$ yields $\mathsf{x}[1]\cdot\mathsf{x}[2]\cdot\mathsf{x}[3]\cdot a \doteq a\cdot\mathsf{y}[1]\cdot\mathsf{y}[2]\cdot\mathsf{y}[3]$. It induces the three equivalence classes $\{\mathsf{x}[1], \mathsf{y}[3], a\}$, $\{\mathsf{x}[2], \mathsf{y}[1]\}$, and $\{\mathsf{x}[3], \mathsf{y}[2]\}$. Every l-substitution h that satisfies $h(\mathsf{x})[1] = h(\mathsf{y})[3] = a$, $h(\mathsf{x})[2] = h(\mathsf{y})[1]$, and $h(\mathsf{x})[3] = h(\mathsf{y})[2]$ is a solution.

It's a well-known result that any satisfiable word equation has a solution in the alphabet $\Sigma(\alpha \doteq \beta)$ [15]. A similar result holds for regular constraints [20] and both results can be combined to show that a word equation with regular constraints on the variables is satisfiable if and only if it has a solution that uses only the constants occurring in the problem (if there is at least one). This no longer holds when negations are allowed. For example, consider the formula $\mathsf{x}\cdot a \doteq a\cdot\mathsf{y} \wedge \mathsf{x} \not\doteq \mathsf{y}$. Any solution h must satisfy $|h(\mathsf{x})| = |h(\mathsf{y})|$ but constructing an h with $h(\mathsf{x}) \neq h(\mathsf{y})$ is not possible if we use only a, the sole constant.

We first generalize the result to formulas of the form $\psi := \alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$, where $\psi_{\neq}$ and $\psi_{\in}$ are conjunctions of inequalities between variables $\mathsf{x} \not\doteq \mathsf{y}$, and regular constraints $\mathsf{x} \mathrel{\dot\in} R$, respectively. This restriction implicitly includes negated regular constraints, as $\mathsf{x} \mathrel{\dot\notin} R$ can be equivalently formulated as $\mathsf{x} \mathrel{\dot\in} \bar{R}$, where $\bar{R}$ is the regular complement of R. For an l-solution h, we define the graph $\mathcal{G}_\psi(h) = (V, E)$ where V is the set of equivalence classes induced by l on $\alpha \doteq \beta$. The set E includes an edge $\{[\mathsf{x}[k]]_\sim, [\mathsf{y}[k]]_\sim\}$ iff $\psi_{\neq}$ contains $\mathsf{x} \not\doteq \mathsf{y}$, $|h(\mathsf{x})| = |h(\mathsf{y})|$, $h(\mathsf{x})[k] \neq h(\mathsf{y})[k]$, and $h(\mathsf{x})[k'] = h(\mathsf{y})[k']$ for all $k' < k$, meaning k is the smallest index where $h(\mathsf{x})$ and $h(\mathsf{y})$ disagree.

If $\mathcal{G}_\psi(h)$ can be colored with n colors, then a new model h' can be constructed using no more than n constants in addition to $\Sigma(\psi)$. Here, a color acts as a new constant, with h' mapping each equivalence class to the vertex color if the original model h mapped its members to a symbol that is not in $\Sigma(\psi)$.

Lemma 1. *Let $\psi := \alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$ and h be a solution. If $\mathcal{G}_\psi(h)$ is n-colorable, then ψ has a solution over $\Sigma(\psi) \cup A$ where A is an alphabet disjoint from $\Sigma(\psi)$ with $|A| = n$.*

Thus, the minimal number of characters required in addition to $\Sigma(\psi)$ is the chromatic number of $\mathcal{G}_\psi(h)$. The next lemma gives an upper bound on this number. This follows from the fact that the graph has at most $|\mathrm{atoms}(\psi_{\neq})|$ edges.

Lemma 2. *Let $\psi := \alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$ be a formula and h be a solution. Then $\mathcal{G}_\psi(h)$ is $|\mathrm{atoms}(\psi_{\neq})| + 1$ colorable.*

Combining the above results gives our main theorem.

Theorem 3. *Let ψ be a formula over word equations and regular constraints with n inequalities. Then ψ has a solution if and only if it has a solution in $\Sigma(\psi) \cup A$ where $A \cap \Sigma(\psi) = \emptyset$ and $|A| = n + 1$.*

The theorem can be shown by assuming ψ is in disjunctive normal form and equivalently rewriting each disjunct to match the form $\alpha \doteq \beta \wedge \psi_{\neq} \wedge \psi_{\in}$. Then, each disjunct can have at most n inequalities, and we can apply Lemma 1 and Lemma 2 to obtain the bound.

The bound on $|A|$ is not always tight because Lemma 2 gives only a coarse bound on the chromatic number. For instance, by applying results from [9], it can be lowered to $\mathcal{O}(\sqrt{n})$ for a formula with n inequalities. However, it is small enough to be practical. Additionally, since formulas often contain fewer inequalities than variables, this result improves our previously known bound of $|\text{vars}(\psi)|$ for formulas ψ that do not include concatenation.

V. Encoding Word Equations

For a first-order formula ψ, our decision procedure fixes some bounds b and translates ψ into a propositional formula $[\![\psi]\!]^b$, which is satisfiable if and only if ψ has a solution within bounds b. The formula is constructed by encoding all literals of ψ individually. We present the encoding for literals that are word equations $\alpha \doteq \beta$, denoted $[\![\alpha \doteq \beta]\!]^b$. Intuitively, $[\![\alpha \doteq \beta]\!]^b$ asks the SAT solver to "guess" a word w for which there exists a substitution h such that $h(\alpha) = w$ and $h(\beta) = w$, making w a *solution word*. It is the conjunction of four formulas $[\![w]\!]^b \wedge [\![lh]\!]^b \wedge [\![m(\alpha)]\!]^b \wedge [\![m(\beta)]\!]^b$, modeling the set of all potential solution words, the set of all possible l-substitutions, and the constraint that the encoded substitution must map both patterns to the same word, respectively. The encoding is sound in the following sense.

Theorem 4. *Let $\alpha \doteq \beta$ be a word equation and b be a function assigning an upper bound to every variable in the equation. Then $\alpha \doteq \beta$ is satisfiable under b if and only if $[\![\alpha \doteq \beta]\!]^b$ is satisfiable.*

If ψ, containing $\alpha \doteq \beta$ as a literal, is not satisfiable with bounds b, then NFA2SAT proceeds to check the satisfiability for larger bounds b'. This results in n calls to the SAT solver, with bounds $b_1, \ldots, b_n$. To make this procedure efficient, the encoding is *incremental*. That is, the encoding $[\![\psi]\!]^{b_k}$ is constructed by only adding more clauses to the formula $[\![\psi]\!]^{b_{k-1}}$. In the following, we present the encoding $[\![\alpha \doteq \beta]\!]^{b_k}$ assuming that $[\![\alpha \doteq \beta]\!]^{b_{k-1}}$ was already encoded. To avoid treating edge cases, we assume $b_0(\mathsf{x}) = 0$ for all $\mathsf{x} \in \Gamma$. Additionally, we assume $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$.

A. Encoding Words

We encode the set of all words that are possible solutions to the equation in $[\![w]\!]^{b_k}$. This includes all words over Σ with length no longer than $U_k = \min(|\overrightarrow{\alpha}^{b_k}|, |\overrightarrow{\beta}^{b_k}|)$, i.e., the length of the smaller of the longest words that either side of the equation can be mapped to under bounds b_k. No substitution h with $|h(\alpha)| > U_k$ or $|h(\beta)| > U_k$ can be a solution w.r.t. to b_k because at least one side of the equation cannot be mapped to a word of length greater than U_k under b_k.

We first pick a new symbol λ that is not in Σ and set $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. The symbol λ denotes an unused position, i.e., a position that is to be mapped to the empty word. Setting an appropriate set of positions to λ allows us to encode all possible words over Σ with length at most U_k. We encode the set $\{\, w \in \Sigma^* \mid |w| \leq U_k \,\}$ by introducing the Boolean variables w_i^c for each position $1 \leq i \leq U_k$ and character $c \in \Sigma_\lambda$. Boolean variable w_i^c is true if c occurs at position i in w. We enforce that exactly one of the w_i^c is true using the following formula

$$[\![w]\!]^{b_k} = \bigwedge_{i=U_{k-1}}^{U_k} EO\{w_i^c \mid c \in \Sigma_\lambda\} \wedge \bigwedge_{i=U_{k-1}}^{U_k - 1} w_i^\lambda \rightarrow w_{i+1}^\lambda .$$

This takes into account that words of length up to U_{k-1} have been encoded in a previous call. In this formula, EO is an encoding of the *exactly-one* constraint on the variables (see [17]). Because concatenation with λ is neutral, we use the second conjunct of the encoding to break symmetry. This ensures that every Boolean assignment σ with $\sigma \models [\![w]\!]^{b_k}$ encodes exactly one word and for every word no longer than U_k there is exactly one σ with $\sigma \models [\![w]\!]^{b_k}$.

B. Encoding l-Substitutions

For all length assignments l bounded by b_k, i.e., for all l with $l(\mathsf{x}) \leq b_k(\mathsf{x})$ for all $\mathsf{x} \in \Gamma$, we encode the set of all possible l-substitutions. This is achieved by initially encoding all substitutions $[\![h]\!]^{b_k}$ and all length assignments in $[\![l]\!]^{b_k}$ (both limited by b_k), and then ensuring that the length of a variable substitution coincides with the length assignment.

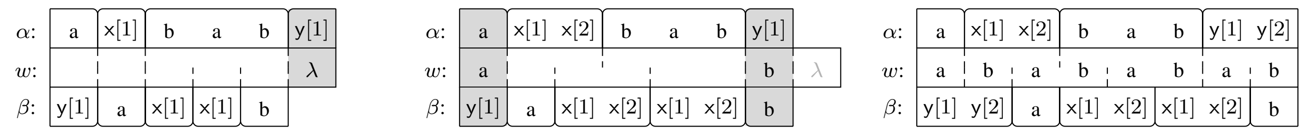
The encoding of substitutions is constructed using a set of Boolean variables $\{\, h_{\mathsf{x}[i]}^a \mid a \in \Sigma_\lambda \,\}$. We ensure that every satisfying assignment to $[\![h]\!]^{b_k}$ encodes exactly one constant word for every variable x, i.e., the substitution of x, by employing an exactly-one constraint exactly as done for the encoding of words.

To encode all possible length assignments, we introduce a set of Boolean variables $\{\, L_{\mathsf{x}}^i \mid 0 \leq i \leq b_k(\mathsf{x}) \,\}$ for all $\mathsf{x} \in \Gamma$. We encode that L_{x}^i is true iff $l(\mathsf{x}) = i$, taking into account that the length assignments for bounds b_{k-1} are already encoded. This makes standard *exactly-one* encodings unsuitable and we instead use the following formula

$$[\![l(\mathsf{x})]\!]^{b_k} := \quad (a_{\mathsf{x},k} \rightarrow \bigvee_{i=b_{k-1}(\mathsf{x})+1}^{b_k(\mathsf{x})} L_{\mathsf{x}}^i \vee a_{\mathsf{x},k-1}) \tag{1}$$

$$\wedge \bigwedge_{i=0}^{b_{k-1}(\mathsf{x})} \bigwedge_{j=b_{k-1}(\mathsf{x})+1}^{b_k(\mathsf{x})} L_{\mathsf{x}}^i \rightarrow \neg L_{\mathsf{x}}^j \tag{2}$$

$$\wedge \bigwedge_{i=b_{k-1}(\mathsf{x})+1}^{b_k(\mathsf{x})} \bigwedge_{j=i+1}^{b_k(\mathsf{x})} L_{\mathsf{x}}^i \rightarrow \neg L_{\mathsf{x}}^j \tag{3}$$



α: a x[1] b a b y[1]
w: λ
β: y[1] a x[1] x[1] b
α: a x[1] x[2] b a b y[1]
w: a b λ
β: y[1] a x[1] x[2] x[1] x[2] b
α: a x[1] x[2] b a b y[1] y[2]
w: a b a b a b a b
β: y[1] y[2] a x[1] x[2] x[1] x[2] b

(a) We first assume an upper bound of 1 for both variables, i.e. $b_1(\mathsf{x}) = b_1(\mathsf{y}) = 1$. In that case, $[\![w]\!]^{b_1}$ encodes all words up to length 6. When assigning length 1 to all variables, $[\![lh]\!]^{b_1}$ and $[\![m(\beta)]\!]^{b_1}$ are conflicting: $[\![lh]\!]$ requires that y[1] is not λ, but since the last segment of β ends at position 5, $[\![m(\beta)]\!]$ requires that the 6th position, which aligns with y[1], is λ. Any other length assignment under b_1 will result in a similar situation. Both, the equation and the encoding are unsatisfiable under b_1.

(b) When assuming an upper bound of 2 for both variables, $b_2(\mathsf{x}) = b_2(\mathsf{y}) = 2$, $[\![w]\!]^{b_2}$ encodes all words up to length 8. Assigning length 2 to x and y maps both patterns to the same length. However, this length assignment still results in a conflict: The first segment of α (a) and the first segment of β (y) both start at position 1, so $[\![m(\alpha)]\!]^{b_2}$ and $[\![m(\beta)]\!]^{b_2}$ entail $w[1] = \mathsf{y}[1] = a$. At the same time, $[\![m(\alpha)]\!]^{b_2}$ and $[\![m(\beta)]\!]^{b_2}$ entail $w[1] = \mathsf{y}[1] = b$ because the respective last segments, b and y, start at position 7.

(c) When instead assigning length 2 to x y, the encoding, and therefore the equation, becomes satisfiable and we find a solution h with $h(\mathsf{x}) = ba$ and $h(\mathsf{y}) = ab$, resulting in the solution word $abababab$.

Fig. 3: Demonstrates of the encoding for the example equation $a \cdot \mathsf{x} \cdot bab \cdot \mathsf{y} \doteq \mathsf{y} \cdot a\mathsf{xx} \cdot b$. The figures illustrate how the encoding operates by fixing length assignments for the variables. This is analogous to the SAT solver assigning truth values to the L^i_{x} variables during the search procedure.

The Boolean variable $a_{\mathsf{x},k}$ is an assumption in the kth call to the SAT solver. Part (1) states that if $a_{\mathsf{x},k}$ is true, then at least one of $\{ L^i_{\mathsf{x}} \mid b_{k-1}(\mathsf{x}) < i \leq b_k(\mathsf{x}) \}$ needs to be true, unless $a_{\mathsf{x},k-1}$ is true (defining $a_{\mathsf{x},0} = \bot$). If $a_{\mathsf{x},k-1}$ is true, then one L^i_{x} with $0 \leq i \leq b_{k-1}(\mathsf{x})$ must be true, establishing that there is at least one $i \leq b_k(\mathsf{x})$ such that L^i_{x} is true. The conjunction (2) and (3) guarantees that at most one L^i_{x} with $i \leq b_k(\mathsf{x})$ is true. Thus, the encoding ensures that exactly one of the variables L^i_{x} with $0 \leq i \leq b_k(\mathsf{x})$ is true.

Finally, $[\![lh]\!]^{b_k}$ combines $[\![h]\!]^{b_k}$ and $[\![l]\!]^{b_k}$ to ensure that the length of each substitution matches the assigned length. This is expressed by ensuring $h(\mathsf{x})[i] \cdots h(\mathsf{x})[b_k(\mathsf{x})] = \varepsilon$ if and only if L^i_{x} is true, using

$$[\![lh]\!]^{b_k} = [\![h]\!]^{b_k} \wedge [\![l]\!]^{b_k} \wedge \bigwedge_{\mathsf{x}\in\Gamma} \bigwedge_{i=b_k(\mathsf{x})}^{b_k(\mathsf{x})-1} (h^{\lambda}_{\mathsf{x}[i+1]} \leftrightarrow L^i_{\mathsf{x}}).$$

Assigning true to $h^{\lambda}_{\mathsf{x}[i+1]}$ encodes that the suffix of the substitution of x starting at $i+1$ is empty. Because exactly one L^i_{x} is true, this asserts that the length of the substitution of x is exactly i if L^i_{x} is true.

C. Matching Patterns To Words

We constrain that any assignment satisfying $[\![lh]\!]^{b_k} \wedge [\![w]\!]^{b_k}$ encodes an l-substitution h and a word w such that $h(\alpha) = w = h(\beta)$, asserting that h is a solution. This is achieved through the formulas $[\![m(\alpha)]\!]^{b_k}$ and $[\![m(\beta)]\!]^{b_k}$, which encode that h maps the ith position of $\overrightarrow{\alpha}^l$ and $\overrightarrow{\beta}^l$ to the ith position of w, for any encode length assignment l. Since the encoding is the same for both sides of the equation, we describe it using a generic pattern γ.

The idea of $[\![m(\gamma)]\!]^{b_k}$ is to split γ into consecutive factors of variables and constant words and assert that if a factor *starts* at position p in $\overrightarrow{\gamma}^l$ and has length k, then its substitution must be equal to the factor of the solution word w from p to $p+k-1$. An example of how this idea is reflected in the encoding is shown in Figure 3.

Formally, we define the segmentation of γ, denoted $\mathrm{seg}(\gamma)$, as the unique factorization $(\gamma_{(1)}, \dots, \gamma_{(n)})$ of γ with $\gamma_{(i)} \in \Gamma$ or $\gamma_{(i)} \in \Sigma^+$ for all $i \leq n$, and if $\gamma_{(i)} \in \Sigma^+$ then either $\gamma_{(i+1)} \in \Gamma$ or $i = n$. For example, the pattern $\mathsf{x}{\cdot}abc{\cdot}\mathsf{y}{\cdot}\mathsf{x}{\cdot}def$ is factorized into five segments $(\mathsf{x}, abc, \mathsf{y}, \mathsf{x}, def)$. Given a length assignment l, the *start position* of $\gamma_{(i)}$ is given by $\sum_{j=1}^{i-1} |\gamma_{(j)}| + 1$, the sum of the lengths of all preceding segments plus one, where $|\gamma_{(i)}|$ is $l(\mathsf{x})$ if $\gamma_{(i)} = \mathsf{x}$ and $|v|$ if $\gamma_{(i)} = v \in \Sigma^+$. The start position of the first segment is thus always 1.

To ensure the matching between the patterns and the solution word, we first encode set set of all possible start positions for each segment of $\mathrm{seg}(\gamma)$ w.r.t. b_k and condition them on the lengths assigned to the variables using the L^i_{x} variables. The encoding then ensures if a segment starts at position p and has length k, the factor of the solution word from p to $p+k-1$ must be equal to $h(\gamma_{(i)})$, the constant word that the encoded morphism h maps $\gamma_{(i)}$ to. The idea is illustrated in Figure 4.

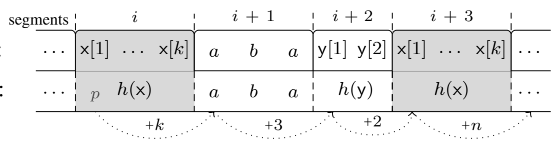


Fig. 4: Matching a pattern γ to a word w. If the ith segment of γ, x, starts at position p, then the factor of w from p to $p+k-1$ must be equal to $h(\mathsf{x})$, and the $i+1$th segment must start at position $p+k$.

To encode the start positions, we introduce a set of Boolean variables $\{ S(\gamma)^p_i \mid 0 \leq p \leq U_k \}$ for all $1 \leq i \leq |\mathrm{seg}(\gamma)|+1$, modeling that $S(\gamma)^p_i$ is true if $\gamma_{(i)}$ starts at position p, where

$S(\gamma)^p_{|\mathrm{seg}(\gamma)|+1}$ marks the end of the pattern. The matching is then encoded using the formula

$$\begin{aligned}[\![m(\gamma)]\!]^{b_k} := \quad & S(\gamma)^1_1 && (1)\\ \wedge & \bigwedge_{p=U_{k-1}}^{U_k} S(\gamma)^p_{|\mathrm{seg}(\gamma)|+1} \to w^\lambda_p && (2)\\ \wedge & \bigwedge_{i=1}^{|\mathrm{seg}(\gamma)|-1} \mathrm{match}(\gamma, i). && (3)\end{aligned}$$

Here, (1) encodes that the first segment starts at the first position. The second part, (2), ensures that the length of the solution word w equals the length of γ under l, by encoding that the first position of w following the last segment is mapped to λ. The last part, (3), establishes the matching between each segment $\gamma_{(i)}$ and the corresponding factor of the solution word, and determines the start position of $\gamma_{(i+1)}$. The encoding depends on whether $\gamma_{(i)}$ is a constant or a variable.

If $\gamma_{(i)} = v$ for some $v \in \Sigma^+$, then $\mathrm{match}(\gamma, i)$ is given by

$$\bigwedge_{p=\max(U_{k-1}-|v|,0)+1}^{U_k-|v|} S(\gamma)^p_i \to \bigwedge_{j=1}^{|v|} w^{v[j]}_{p+j-1} \wedge S^{p+|v|}_{i+1}.$$

The formula states that if $\gamma_{(i)}$ starts at position p, then the factor of w from p to $p + |v| - 1$ must be equal to v and $\gamma_{(i+1)}$ starts at $p + |v|$. The latest position at which the $\gamma_{(i)}$ can start is $U_k - |v|$, as otherwise, it would exceed U_k.

If $\gamma_{(i)} = \mathsf{x}$ for some variable $\mathsf{x} \in \Gamma$, then $\mathrm{match}(\gamma, i)$ is instead given by

$$\bigwedge_{(p,l)\in M_k\setminus M_{k-1}} S(\gamma)^p_i \wedge L^l_{\mathsf{x}} \to \bigwedge_{j=0}^{l-1} \bigwedge_{c\in\Sigma} (h^c_{\mathsf{x}[j]} \leftrightarrow w^c_{p+j}) \wedge S^{p+l}_{i+1}.$$

Here, $M_k = \{ (p, l) \mid p < U_k \wedge l \leq b_k(\mathsf{x}) \wedge p + l \leq U_k \}$ is the set of all pairs of positions and length assignments w.r.t b_k, such that $p + l(\mathsf{x}) \leq U_k$. The formula ensures that if $\gamma_{(i)}$ starts at position p and has length l, then the factor of w from p to $p + l - 1$ must be equal to $h(\mathsf{x})$, and that $\gamma_{(i+1)}$ starts at position $p + l$.

To guide the SAT solver, we impose an *at-most-one* constraint on $\{ S(\gamma)^p_i \mid 0 \leq p \leq U_k \}$ for all $1 \leq i \leq |\mathrm{seg}(\gamma)| + 1$. Additionally, we *disable* all infeasible start positions relative to U_k with assumptions. For segments $\gamma_{(i)} = v \in \Sigma^+$, we add $\neg S(\gamma)^p_i$ as an assumption for all p with $p > U_k - |v|$. For segments $\gamma_{(i)} = \mathsf{x} \in \Gamma$, we add the clauses $a \to \neg(L^l_{\mathsf{x}} \wedge S(\gamma)^p_i)$, with fresh variable a, for all l, p with $l < b(\mathsf{x})$, $p < U_k$, and $l + p > U_k$, and add a as an assumption.

VI. Finding and Refining Bounds

Whenever the SAT solver determines that the formula is unsatisfiable under bounds b, our procedure continues with larger bounds b'. This terminates once the bounds are either large enough to construct a solution or exceed the bounds of the smallest model, as explained in Section III.

Theoretical bounds on the minimal solution to a word equation can be computed, but these bounds can be doubly exponential [24]. Given this complexity, using exact minimal bounds is impractical. Instead, we employ a heuristic to identify tighter bounds by extracting linear constraints on variable lengths from the word equations. We use a known method for bounding the solutions of the resulting linear integer problem. This approach is sound but not complete as it may fail to find finite bounds on the variables.

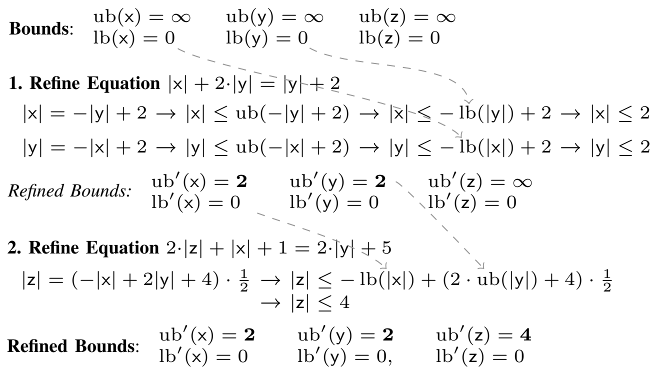
Algorithm 1 Iterative Bound Refinement

```
Input: Conjunctive formula $\psi$
Output: Bounds for $\psi$ or UNSAT if $\psi$ is unsatisfiable
  $\mathrm{lb}, \mathrm{ub} \leftarrow \mathrm{init}(\psi)$
  repeat
      for $\alpha \doteq \beta \in \mathrm{atoms}(\psi)$ do
          $\mathrm{lb}', \mathrm{ub}' \leftarrow$ refinement_step($[\alpha \doteq \beta]_L$, lb, ub)
          if conflict(lb', ub') then
              return UNSAT
          end if
      end for
  until $\mathrm{lb}' = \mathrm{lb} \wedge \mathrm{ub}' = \mathrm{ub}$
  return $\mathrm{lb}, \mathrm{ub}$
```

For a word equation $\alpha \doteq \beta$ we define the linear integer equation $[\alpha \doteq \beta]_L := \sum_{\mathsf{x}\in\Gamma}(|\alpha|_{\mathsf{x}} - |\beta|_{\mathsf{x}}) \cdot |\mathsf{x}| = \sum_{a\in\Sigma} |\beta|_a - |\alpha|_a$. The equation is satisfied by every substitution h with $|h(\alpha)| = |h(\beta)|$. Especially, if h is a solution for $\alpha \doteq \beta$, then it also satisfies $[\alpha \doteq \beta]_L$. Conversely, if a h does not satisfy $[\alpha \doteq \beta]_L$, it is not a solution for $\alpha \doteq \beta$. For instance, consider the word equation $\mathsf{z}{\cdot}b{\cdot}\mathsf{z}{\cdot}\mathsf{x} \doteq ba{\cdot}\mathsf{y}{\cdot}a{\cdot}\mathsf{y}{\cdot}bb$ which has the corresponding integer equation $2{\cdot}|\mathsf{z}| + |\mathsf{x}| + 1 = 2{\cdot}|\mathsf{y}| + 5$. The substitution $h = \{\mathsf{x} \leftarrow a, \mathsf{y} \leftarrow a, \mathsf{z} \leftarrow baaaa\}$ is not a solution because $2|h(\mathsf{z})| + |h(\mathsf{x})| + 1 = 12 \neq 9 = 2|h(\mathsf{y})| + 5$.

We treat $[\alpha \doteq \beta]_L$ as an equation over variables $\Gamma_L = \{ |\mathsf{x}| \mid \mathsf{x} \in \Gamma \}$. For a conjunction of word equations ψ, we lift $[\psi]_L$ to the conjunction of the corresponding integer equations. Our procedure computes lower and upper bounds $\mathrm{lb}, \mathrm{ub} : \Gamma_L \to \mathbb{N} \cup \infty$ such that $\mathrm{lb}(|\mathsf{x}|) \leq g(|\mathsf{x}|) \leq \mathrm{ub}(|\mathsf{x}|)$ for all $g : \Gamma_L \to \mathbb{N}$ satisfying $[\psi]_L$. The algorithm is sketched in Algorithm 1 and an example is shown in Figure 5.

Iterative Bound Refinement: For a conjunctive formula ψ in normal form, we derive bounds on the variables of ψ using an incremental refinement procedure. If variable x is constrained to belong to a regular expression R and R is recognized by a cycle-free n-state automaton then we set the initial bounds on x to $\mathrm{lb}_0(\mathsf{x}) = 0$ and $\mathrm{ub}_0(\mathsf{x}) = n$. Otherwise, we initialize the bounds to $\mathrm{lb}_0(\mathsf{x}) = 0$ and $\mathrm{ub}_0(\mathsf{x}) = \infty$. Thus, $\mathrm{lb}_0(|\mathsf{x}|) \leq g(|\mathsf{x}|) \leq \mathrm{ub}_0(|\mathsf{x}|)$ holds initially. The algorithm then iteratively refines the bounds until a conflict is detected or a fixed point is reached. A conflict occurs if $\mathrm{lb}(|\mathsf{x}|) > \mathrm{ub}(|\mathsf{x}|)$, $\mathrm{ub}(|\mathsf{x}|) < 0$ or $\mathrm{lb}(|\mathsf{x}|) = \infty$ for some variable x, in which case $[\psi]_L$, and therefore ψ, are unsatisfiable. If no conflict is found, then the functions lb and ub provide bounds on the lengths of the variables for the solutions to ψ. Specifically, if $\mathrm{ub}(|\mathsf{x}|) \neq \infty$ for all x, then ub are bounds on the smallest model for ψ. Algorithm 1 may not terminate in general (see [13]). In our implementation, we enforce an upper limit on the number of iterations and we return the best bounds available when this limit is reached.



Bounds: $\mathrm{ub}(\mathsf{x}) = \infty$ $\mathrm{ub}(\mathsf{y}) = \infty$ $\mathrm{ub}(\mathsf{z}) = \infty$
$\mathrm{lb}(\mathsf{x}) = 0$ $\mathrm{lb}(\mathsf{y}) = 0$ $\mathrm{lb}(\mathsf{z}) = 0$

1. Refine Equation $|\mathsf{x}| + 2{\cdot}|\mathsf{y}| = |\mathsf{y}| + 2$

$$|\mathsf{x}| = -|\mathsf{y}| + 2 \rightarrow |\mathsf{x}| \leq \mathrm{ub}(-|\mathsf{y}| + 2) \rightarrow |\mathsf{x}| \leq -\,\mathrm{lb}(|\mathsf{y}|) + 2 \rightarrow |\mathsf{x}| \leq 2$$

$$|\mathsf{y}| = -|\mathsf{x}| + 2 \rightarrow |\mathsf{y}| \leq \mathrm{ub}(-|\mathsf{x}| + 2) \rightarrow |\mathsf{y}| \leq -\,\mathrm{lb}(|\mathsf{x}|) + 2 \rightarrow |\mathsf{y}| \leq 2$$

Refined Bounds: $\mathrm{ub}'(\mathsf{x}) = \mathbf{2}$ $\mathrm{ub}'(\mathsf{y}) = \mathbf{2}$ $\mathrm{ub}'(\mathsf{z}) = \infty$
$\mathrm{lb}'(\mathsf{x}) = 0$ $\mathrm{lb}'(\mathsf{y}) = 0$ $\mathrm{lb}'(\mathsf{z}) = 0$

2. Refine Equation $2{\cdot}|\mathsf{z}| + |\mathsf{x}| + 1 = 2{\cdot}|\mathsf{y}| + 5$

$$|\mathsf{z}| = (-|\mathsf{x}| + 2|\mathsf{y}| + 4) \cdot \tfrac{1}{2} \rightarrow |\mathsf{z}| \leq -\,\mathrm{lb}(|\mathsf{x}|) + (2 \cdot \mathrm{ub}(|\mathsf{y}|) + 4) \cdot \tfrac{1}{2}$$
$$\rightarrow |\mathsf{z}| \leq 4$$

Refined Bounds: $\mathrm{ub}'(\mathsf{x}) = \mathbf{2}$ $\mathrm{ub}'(\mathsf{y}) = \mathbf{2}$ $\mathrm{ub}'(\mathsf{z}) = \mathbf{4}$
$\mathrm{lb}'(\mathsf{x}) = 0$ $\mathrm{lb}'(\mathsf{y}) = 0,$ $\mathrm{lb}'(\mathsf{z}) = 0$

Fig. 5: Bound refinement for the system of word equations $\mathsf{x}{\cdot}\mathsf{y}{\cdot}\mathsf{y} \doteq a{\cdot}\mathsf{y}{\cdot}a \wedge \mathsf{z}{\cdot}b{\cdot}\mathsf{z}{\cdot}\mathsf{x} \doteq ba{\cdot}\mathsf{y}{\cdot}a{\cdot}\mathsf{y}{\cdot}bb$. Only upper bounds are shown. The initial bounds on $|\mathsf{x}|$, $|\mathsf{y}|$, and $|\mathsf{z}|$ are $[0, \infty)$. After processing the two equations, the bounds are refined to $0 \leq |\mathsf{x}| \leq 2$, $0 \leq |\mathsf{y}| \leq 2$, and $0 \leq |\mathsf{z}| \leq 4$.

Refinement Steps: The key part in the procedure is the *implied bound refinement* of [13]. This procedure extends the functions lb, ub to arithmetic terms. For constants c, set $\mathrm{lb}(c) = \mathrm{ub}(c) = c$. For terms of the form $c|\mathsf{x}|$, the value depends on whether the constant c is positive. If $c \geq 0$, then $\mathrm{lb}(c|\mathsf{x}|) = c \cdot \mathrm{lb}(|\mathsf{x}|)$ and $\mathrm{ub}(c|\mathsf{x}|) = c \cdot \mathrm{ub}(|\mathsf{x}|)$. If $c < 0$, then $\mathrm{lb}(c|\mathsf{x}|) = c{\cdot}\mathrm{ub}(|\mathsf{x}|)$ and $\mathrm{ub}(c|\mathsf{x}|) = c{\cdot}\mathrm{lb}(|\mathsf{x}|)$ instead. For sums of the form $T_1 + T_2$, we just use $\mathrm{lb}(T_1 + T_2) = \mathrm{lb}(T_1) + \mathrm{lb}(T_2)$ and $\mathrm{ub}(T_1 + T_2) = \mathrm{ub}(T_1) + \mathrm{ub}(T_2)$. Then, for any term T, $\mathrm{lb}(T)$ and $\mathrm{ub}(T)$ are the smallest and largest value T can assume when respecting the bounds that lb and ub impose on the variables:

Lemma 5. *Let* $|\mathsf{x}| = T$ *be* $[\alpha \doteq \beta]_L$ *solved for* x, *and* lb *and* ub *be bounds for* $[\alpha \doteq \beta]_L$. *If* g *is a solution for* $[\alpha \doteq \beta]_L$, *then* $\mathrm{lb}(T) \leq g(|\mathsf{x}|) \leq \mathrm{ub}(T)$ *holds.*

If lb and ub are lower and upper bounds on the solutions for $[\alpha \doteq \beta]_L$, and the linear constraints imply an equation of the form $|\mathsf{x}| = T$ where T does not contain $|\mathsf{x}|$, then $\mathrm{lb}(T)$ and $\mathrm{ub}(T)$ are lower and upper bounds for $|\mathsf{x}|$ in $[\alpha \doteq \beta]_L$. If these bounds improve on $\mathrm{lb}(|\mathsf{x}|)$ and $\mathrm{ub}(|\mathsf{x}|)$ then the procedure updates both and iterates.

VII. Experimental Evaluation

The NFA2SAT solver is written in Rust and uses the SAT solver CaDiCaL-1.5.2. The source code consists of about 18k lines of Rust. Compared with the earlier version described in a previous paper [20], we have made several extensions to support word equations. First, the input formula φ is rewritten into an equivalent formula in normal form in which all literals are either word equations, inequalities between variables, or (negated) regular constraints (see Section II). The propositional encoding of regular constraints and inequalities between variables is explained in [20] and has not changed. Word equations are encoded as explained in Section V and the alphabet reduction is implemented as explained in Section IV. The bound refinement technique explained in Section VI is used between SAT solver invocations to obtain small model bounds and handle unsatisfiable instances, if the UNSAT core contains (encoded) word equations.

Some types of negated constraints cannot be encoded directly because they implicitly introduce universal quantifiers. For example, the literal $\neg\,\mathsf{contains}(a{\cdot}\mathsf{x}{\cdot}b, \mathsf{y})$ with variables x and y, and constants a and b, is equivalent to $\forall \mathsf{z}_1\mathsf{z}_2.\ \mathsf{z}_1{\cdot}\mathsf{y}{\cdot}\mathsf{z}_2 \neq a{\cdot}\mathsf{x}{\cdot}b$. We handle such constraints lazily using a CEGAR-style approach: NFA2SAT tries to find a solution that ignores these types of constraints. If a solution is found, the solver checks whether it satisfies the negated constraints that were ignored. If so, the original formula is satisfiable. If some of the unhandled negated constraints are not true, then we restart NFA2SAT with a constraint that forces it to search for another solution.

We compare NFA2SAT with CVC5 (version 1.1.1), Z3 (version 4.13.0), Noodler (commit #e1e46068) and Ostrich (commit #f7f0aa8c). We also include results from NFA2SAT when the bound refinement is disabled. We run the experiments on an Amazon EC2 M5.24xlarge instance running Amazon Linux 2, equipped with 384 GB RAM and 96 Intel Xeon CPUs running at 2.50 GHz. We ran 48 solvers in parallel, with a 300 second timeout and a 16 GB memory limit per problem.
We have evaluated our approach on the ZaligVinder [19] benchmark set[1]. The set contains 82,632 problems from different sources and includes all string problems from SMT-LIB. Out of these problems, 33,091 are in the logical fragment supported by NFA2SAT. The others include constraints currently unsupported by NFA2SAT, e.g., constraints on string lengths.

Table I summarizes the results. The table shows the number of satisfiable and unsatisfiable problems solved by each solver. It also includes the total runtime of each solver on the problems it successfully solves. On these benchmarks, NFA2SAT is competitive with CVC5, Z3, and Ostrich. Noodler is faster overall than the other solvers by a significant margin. The baseline version of NFA2SAT solves more problems in total than CVC5, but fewer than Z3, Ostrich, and Noodler. Noodler solves the most problems overall. The table also shows that the bound refinement heuristic helps performance on both satisfiable and unsatisfiable instances. It increases the number of solved problems by 39 and reduces the total runtime by 6,138 seconds. The table shows that the solvers have different characteristics. CVC5 is faster than the other solvers on satisfiable problems but it is slower on unsatisfiable instances. Conversely, Z3, Ostrich, and both versions of NFA2SAT are slower overall on satisfiable instances. Noodler is the fastest solver on unsatisfiable problems. NFA2SAT comes second on unsatisfiable problems, but solves fewer problems than Noodler. Ostrich is close to Noodler in terms of the number of solved unsatisfiable problems but it is slower. The table also shows that NFA2SAT is faster on average than Z3, Ostrich, and CVC5 on the problems that it can solve, only Noodler is faster.

[1]Available at https://github.com/zaligvinder/zaligvinder

TABLE I: Results on the ZaligVinder Benchmarks. NFA2SAT is our baseline solver. NFA2SAT (no ref) is the same solver with bound refinement disabled.

	Solved Problems			Runtimes Total (s)			Runtimes Average (s)		
Solver	Sat	Unsat	Total	Sat	Unsat	Total	Sat	Unsat	Total
CVC5-1.1.1	25,443	7,058	32,501	1,250.40	34,285.09	35,535.49	0.05	4.86	1.09
Z3-4.13.0	25,480	7,164	32,644	43,177.51	3,083.34	46,260.85	1.69	0.43	1.42
ostrich	25,439	7,388	32,827	107,786.97	62,183.89	169,970.86	4.24	8.42	5.18
noodler	25,536	7,539	33,075	1,276.80	407.51	1,684.31	0.05	0.05	0.05
NFA2SAT	25,406	7,118	32,524	15,276.99	1,389.70	17,666.69	0.6	0.2	0.54
NFA2SAT (no ref)	25,384	7,101	32,485	21,624.50	2,180.07	23,804.57	0.85	0.31	0.73

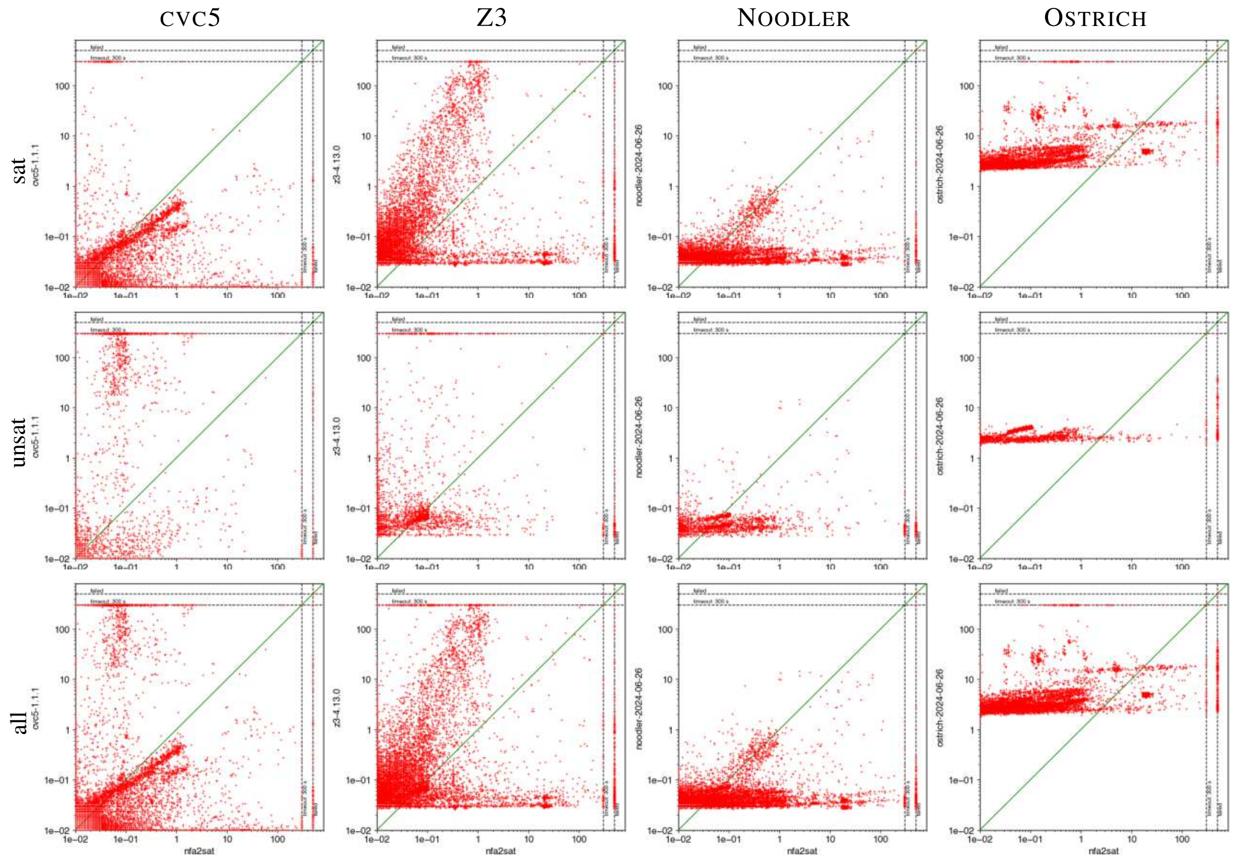


Fig. 6: Scatter plots comparing NFA2SAT (x-axis) with CVC5, Z3, NOODLER, and OSTRICH (y-axis). The first row contains only satisfiable, the second row only unsatisfiable, and the last row all problems. The axes are on a logarithmic scale. The diagonal line represents equal runtime. Points above the diagonal are problems where NFA2SAT is faster. The first dashed line represents timeouts. The second dashed line represents failures (crashes, out-of-memory).

The scatter plots in Figure 6 show that the techniques employed by NFA2SAT and the other solvers are complementary. Every column compares NFA2SAT with a different solver. The leftmost plots show that CVC5 is generally faster on satisfiable examples (points below the diagonal), but not on all problems. The converse happens on unsatisfiable problems (second row). One can also see that CVC5 and NFA2SAT are good on different sets of unsatisfiable benchmarks: the plot for unsatisfiable problems does not have many points close to the diagonal. This behavior is even more pronounced when we compare NFA2SAT and Z3 (second column). The plots show a pattern where some benchmarks are easier for NFA2SAT and others are easier for Z3, with not many points along the diagonal. Many problems solved by NFA2SAT in less than 1 second are harder for Z3, and conversely, many problems solved by Z3 in less than 0.1 seconds are hard for NFA2SAT. We can also see that Z3 has a higher startup cost than NFA2SAT and CVC5 on these problems. The plots comparing NFA2SAT with NOODLER show a pattern similar to the comparison with CVC5, but NOODLER is faster overall. For satisfiable problems, there are many cases along the diagonal, indicating that the solvers perform equally well on these benchmarks. However, there is also a large set of problems where NOODLER is faster, as shown by the concentration

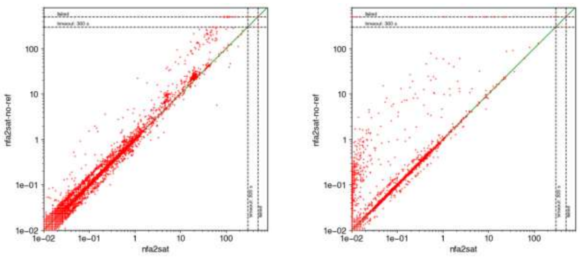


Fig. 7: NFA2SAT with and without bound refinement on SAT and UNSAT instances, respectively.

of points below the diagonal. Despite this, NFA2SAT still outperforms NOODLER on a subset of problems, both satisfiable and unsatisfiable. NOODLER is based on Z3, and thus shares the same startup cost. The plots comparing NFA2SAT with OSTRICH indicate that NFA2SAT is overall faster on most problems, both satisfiable and unsatisfiable, as shown by the majority of points above the diagonal. For many of these problems, NFA2SAT's advantage can be attributed to OSTRICH's high startup time, which is about 2 seconds. There is a significant number of both satisfiable and unsatisfiable problems where OSTRICH is faster, or which OSTRICHsolved but NFA2SAT could not solve. On unsatisfiable problems, OSTRICH solves more problems than NFA2SAT. Only a few instances are close to the diagonal, emphasizing that NFA2SAT and OSTRICH complement each other.

The scatter plots in Figure 7 show the impact of the bound-refinement heuristics. The plot shows many points close to the diagonal, which are problems where bound refinement does not help or hurt. But most of the other points are above the diagonal. These are problems where bound refinement improves runtime.

VIII. Conclusion

We have added support for word equations to the NFA2SAT string solver. Our approach relies on a novel SAT encoding of word equations that is based on enumerating constant words and matching both sides of a word equation to the same constant word. The encoding makes use of incremental SAT solving. To detect unsatisfiable instances, we propose an incomplete but practical technique that derives linear constraints on the length of variables occurring in word equations and uses a bound-refinement algorithm. An empirical evaluation on a large set of benchmarks demonstrates that our approach is competitive with the state-of-the-art solvers CVC5 and Z3. More important, the techniques employed by NFA2SAT are complementary which brings benefits to portfolio-solving strategies. In future work, we plan to support atoms that constrain the lengths of strings. Additionally, we want to explore a more diverse array of approaches to determine unsatisfiability and optimize the SAT encoding in order to improve the solver's efficiency.

References

[1] Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8602994

[2] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 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. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24

[3] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`

[4] Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: String theories involving regular membership predicates: From practice to theory and back. In: Lecroq, T., Puzynina, S. (eds.) Combinatorics on Words. pp. 50–64. Springer International Publishing, Cham (2021)

[5] Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498707, https://doi.org/10.1145/3498707

[6] Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 24–33. Springer Nature Switzerland, Cham (2024)

[7] Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On Solving Word Equations Using SAT, p. 93–106. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-30806-3_8

[8] Diekert, V.: Makanin's algorithm for solving word equations with regular constraints. Tech. Rep. 1998/02, University of Stuttgart (March 1998). https://doi.org/10.18419/opus-2419, https://elib.uni-stuttgart.de/bitstream/11682/2436/1/420_1.pdf

[9] Diestel, R.: Graph Theory, Graduate Texts in Mathematics, vol. 173. Springer (1997)

[10] Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003). https://doi.org/10.1016/S1571-0661(05)82542-3, bMC'2003, First International Workshop on Bounded Model Checking

[11] Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) Programming Languages and Systems. pp. 19–30. Springer International Publishing, Cham (2019)

[12] Jez, A.: Word Equations in Nondeterministic Linear Space. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 80, pp. 95:1–95:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.95

[13] Jovanović, D., de Moura, L.: Cutting to the Chase Solving Linear Integer Arithmetic, pp. 338–353. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_26

[14] Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: CertiStr: a certified string solver. In: Popescu, A., Zdancewic, S. (eds.) CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022. pp. 210–224. ACM (2022). https://doi.org/10.1145/3497775.3503691

[15] Karhumäki, J., Mignosi, F., Plandowski, W.: The expressibility of languages and relations by word equations. J. ACM **47**(3), 483–505 (may 2000). https://doi.org/10.1145/337244.337255

[16] Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. p. 105–116. ISSTA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572286

[17] Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from N objects. In: Fourth Workshop on Constraints in Formal Verification (CFV) (2007)

[18] Kulczynski, M., Lotz, K., Nowotka, D., Poulsen, D.B.: Solving string theories involving regular membership predicates using SAT. In: Legunsen, O., Rosu, G. (eds.) Model Checking Software. pp. 134–151. Springer International Publishing, Cham (2022)

[19] Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: ZaligVinder: A generic test framework for string solvers. Journal of Software: Evolution and Process **35**(4), e2400 (2023). https://doi.org/https://doi.org/10.1002/smr.2400

[20] Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Majumdar, R., Nowotka, D.: Solving string constraints using SAT. In: Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II. pp. 187–208. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-37703-7_9

[21] Lu, Z., Siemer, S., Jha, P., Manea, F., Day, J., Ganesh, V.: Z3-alpha: a reinforcement learning guided smt solver. System Description: SMT-COMP 2023 (July 2023), https://smt-comp.github.io/2023/system-descriptions/z3-alpha.pdf

[22] Makanin, G.S.: The problem of solvability of equations in a free semigroup. Mathematics of The Ussr-sbornik **32**, 129–198 (1977), https://api.semanticscholar.org/CorpusID:7073856

[23] 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. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)

[24] Plandowski, W.: Satisfiability of word equations with constants is in nexptime. In: Proceedings of the thirty-first annual ACM symposium on Theory of Computing. STOC99, ACM (May 1999). https://doi.org/10.1145/301250.301443

[25] Plandowski, W., Rytter, W.: Application of Lempel-Ziv encodings to the solution of word equations, p. 731–742. Springer Berlin Heidelberg (1998). https://doi.org/10.1007/bfb0055097

[26] Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 3–18. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_1

[27] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: 2010 IEEE Symposium on Security and Privacy. pp. 513–528 (2010). https://doi.org/10.1109/SP.2010.38

[28] Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization, pp. 85–150. Springer Berlin Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4

[29] Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design **44**(1), 44–70 (2014). https://doi.org/10.1007/s10703-013-0189-1

SMT-D: New Strategies for Portfolio-Based SMT Solving

Clark Barrett[1, 5], Pei-Wei Chen[3,*], Byron Cook[1], Bruno Dutertre[1], Robert B. Jones[1], Nham Le[1,2,*], Andrew Reynolds[1,6], Kunal Sheth[4,*], Christopher Stephens[1], and Michael W. Whalen[1]

[1]Amazon Web Services, Seattle, USA, {byron, dutebrun, rbtjones, chrisss, mww}@amazon.com
[2]University of Waterloo, Warterloo, Canada, nham.van.le@uwaterloo.ca
[3]University of California, Berkeley, USA, pwchen@berkeley.edu
[4]University of Illinois, Urbana-Champaign, USA, kunal@kunalsheth.info
[5]Stanford University, Stanford, USA, barrett@cs.stanford.edu
[6]University of Iowa, Iowa City, USA, andrew-reynolds@uiowa.edu

***Abstract*—We introduce SMT-D, a tool for portfolio-based distributed SMT solving. We propose a general architecture consisting of two main components: (i) solvers extended with the capability of sharing and importing information on the fly while solving; and (ii) a central manager that orchestrates and monitors solvers while also deciding which information to share with which solvers. We introduce new information-sharing strategies based on the idea of maximizing the amount of useful diversity in the system. We show that on hard benchmarks from recent related work, SMT-D instantiated with the cvc5 SMT solver achieves significant speed-up over sequential performance, is competitive with existing portfolio approaches, and contributes a number of unique solutions.**

I. Introduction

Solvers for satisfiability modulo theories (SMT) are used as general-purpose constraint solvers in a wide variety of applications, including those arising in computer science [6], [10], mathematics [12], [21], operations research [20], and more. Unsurprisingly, as users push SMT solvers to solve more diverse and challenging problems, solver performance becomes the limiting factor in many applications.

Today, state-of-the-art SMT solvers like cvc5 [2], Yices [8], and Z3 [7] do not benefit from additional cores, and if the solving job times out or crashes, any work done during the solving attempt is lost. An effective strategy for distributed SMT solving could address both issues: it can help scale SMT solving across multiple threads and machines, and by sharing information among solver instances, any progress made can be retained and used by others, even if one of the instances crashes or fails.

Two main approaches to distributed SMT solving have been explored: portfolio solving and divide-and-conquer. Portfolio solving is essentially a race between multiple independent SMT solver instances. Each solver is different in some way: either it is a completely different solver, or it is configured differently, or it is provided with a different (but logically equivalent) input. Portfolio solving aims to leverage the well-known high variance that often exists when solving equivalent SMT problems: the hope is that one of the solvers in the portfolio finishes quickly. Portfolio solving can be enhanced by sharing information among the solver instances. Typically, this information consists of formulas that the SMT solvers have learned that can be used to prune the search space. In divide-and-conquer solving, a single problem is partitioned in such a way that if each partition is solved, this provides a solution to the original problem. The main challenge is finding a way to divide the problem that actually improves performance.

In this paper, we introduce SMT-D, a new tool for portfolio-based distributed SMT solving. SMT-D's architecture consists of two main components: (i) solvers extended with the capability to share and import information on the fly while solving; and (ii) a central manager that orchestrates and monitors solvers while also deciding which information to share with which solvers. We also introduce a new information-sharing strategy based on the idea of maximizing the amount of "good" diversity in the system. On hard benchmarks from recent work [22], SMT-D instantiated with the cvc5 SMT solver achieves significant speed-ups over sequential performance, is competitive with existing portfolio approaches, and contributes a number of unique solutions.

In summary, our contributions include:

- a flexible and general architecture for portfolio-based SMT solving with information sharing;
- new portfolio strategies including *delayed sharing* and *guided randomization*;
- an implementation in SMT-D; and
- an evaluation of SMT-D and existing systems on several sets of challenging benchmarks.

The rest of the paper is organized as follows. Section II covers background and related work. Section III describes the architecture of SMT-D. Section IV explains our novel portfolio strategies, and Section V provides additional implementation details. Experimental results are reported in Section VI, and

*These authors did much of the work on this project and did so during internships at Amazon Web Services.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_10

Algorithm 1: The CDCL(T) loop

```
Input  : an SMT formula $F$
Output: SAT or UNSAT
1  $clauseDB \leftarrow toCNF(F)$;
2  while True do
3      do
4          $conflict \leftarrow BooleanPropagate(clauseDB)$;
5          $changed \leftarrow$ False;
6          if $conflict = \emptyset$ then
7              $conflict, changed \leftarrow theoryCheck()$ ;
8      while $changed \wedge conflict = \emptyset$;
9      if $conflict \neq \emptyset$ then
10         $level, lemma \leftarrow resolveConflict(conflict)$;
11         $clauseDB \leftarrow clauseDB \cup lemma$;
12         if $level < 0$ then
13             return UNSAT;
14         backtrack($level$) ;
15     else
16         if $nextLiteral() = NULL$ then
17             return SAT ;
```

Section VII concludes.

II. Preliminaries

We assume the standard logical setting for SMT with the usual notions of terms, interpretations, and theories (see, e.g., [5]). We assume a fixed background theory $\mathcal{T}$ (which could be a composition of one or more individual theories). A $\mathcal{T}$-*interpretation* is an interpretation that interprets symbols in $\mathcal{T}$ as expected. An *atom* is a term of sort Bool that does not contain any proper sub-terms of sort Bool. A *literal* is either an atom or the negation of an atom. A clause is a disjunction of literals, and a *cube* is a conjunction of literals. A formula is a term of sort Bool and is *satisfiable* (resp., *unsatisfiable*) if it is satisfied by some (resp., no) $\mathcal{T}$-interpretation. A formula whose negation is unsatisfiable is *valid*.

A. CDCL(T)-Based SMT Solvers

Most modern SMT solvers are based on the CDCL(T) framework [17], in which a SAT solver and one or more theory solvers cooperate. The SAT solver incrementally builds a truth assignment for the *Boolean skeleton* of the formula, obtained by replacing each unique atom by a Boolean variable. It does this using a standard CDCL loop that is modified to also take into account theory reasoning. The modified CDCL(T) approach is shown in Algorithm 1. Initially, an input formula F is converted to conjunctive normal form (CNF), and each clause is stored in a clause database. The main loop first calls Boolean propagation, which may assign some atoms to true or false. If Boolean propagation produces no conflicts, then the theory solvers are called to check for theory conflicts. These two steps repeat until a fixed point is reached. If there is a conflict, it is resolved by learning a conflict lemma and backtracking to an earlier level in which there is no conflict. Otherwise, the $nextLiteral$ function is used to make a case split on a new literal. More details can be found in [5].

B. Portfolio Solving with Lemma Sharing

SMT solvers are highly sensitive. Small changes to the input formula or solver heuristics can result in orders of magnitude difference in solving time [11]. While a cause of frustration for users, this phenomenon can be leveraged to create an effective *portfolio* solving strategy: multiple solvers (each configured differently or with permuted, but logically equivalent, inputs) are run in a "racing" mode and the result of the fastest one is returned. This approach has been explored extensively for both SAT and SMT solving [1], [15], [16], [24] and produces reliable speed-ups [23]. Still, portfolio solving is limited by the performance of the best and luckiest individual solver, leading to diminishing returns with increasing parallelism. Additional performance can be obtained with *information sharing*. Each solver in the portfolio shares its learned conflict lemmas with the others, with the hope that this exchange of information will help find the solution faster.

Implementing a lemma-sharing portfolio in practice is highly non-trivial. System-wise, one must provide scalability, fault tolerance, and low overhead; algorithmic-wise, one must find a good balance between sharing useful information and overloading the system with too many lemmas. Moreover, a well-designed distributed solver should be modular and general, leaving room for future extensions. Ideally, it should also accommodate a wide range of different solvers, support new sharing strategies, and be compatible with other parallel strategies such as partitioning. After a review of related work, we discuss our design and implementation, including design decisions that aim to meet the criteria mentioned above.

C. Related Work

Parallel strategies for SAT solving have been explored extensively [1], [13], [14], [24]. SMT solvers must take into account the more sophisticated CDCL(T) architecture and the different performance profiles of SMT applications. However, the two main approaches for parallel SAT solving are also found in the existing research literature on parallel SMT solving, namely *portfolio solving* and *partitioning*.

Portfolio solving for SMT. Z3 was the first SMT solver to implement portfolio solving with information sharing [23]. The Z3 implementation focuses on a shared-memory implementation and achieves a speed-up of 3.5x on average for moderately difficult integer difference logic benchmarks using a portfolio of four copies of Z3. The sharing strategy used is simple: lemmas with eight literals or fewer are shared, and others are not. Shared lemmas are put into a queue, and each solver in the portfolio checks its queue whenever it backtracks to decision level 0. Unfortunately, portfolio solving is no longer supported in recent versions of Z3.

SMTS [15] is another system implementing portfolio solving with information sharing. As with the Z3 approach, lemmas to be shared are loaded into queues that are accessed when the solvers backtrack to decision level 0. SMTS uses a central database to store shared lemmas. A filtering heuristic is used to decide which lemmas to add to the database, and a selection heuristic is used to decide which lemmas to share from the database. SMTS obtains its best results using a filter that discards lemmas with more than four literals and a selection heuristic that randomly samples from the database. The SMTS authors specifically flag the need for better filtering and selection techniques in their discussion of future work. Our work builds on and extends these previous approaches in several ways, as we discuss in the next section.

Partitioning in SMT. SMTS [15] implements several partitioning strategies that outperform sequential solving. Relatedly, Wilson et al. [22] implement a partitioning-based parallel solver using cvc5 (which we will refer to as CVC5-P going forward) and show that it outperforms traditional portfolio solving on a set of challenging benchmarks. CVC5-P does not use any information sharing, leaving the integration of sharing to future work. SMTS does explore a limited form of sharing mixed with partitioning: each partition can be solved using a portfolio with lemma sharing, which yields even better performance. The focus of this paper is on portfolio solving with sharing but without partitioning. We aim to build a robust and high-performance solution that could be expanded to include partitioning strategies in future work.

III. An Architecture for Portfolio-Based SMT Solving

In this section, we describe a general architecture for portfolio-based SMT solving and contrast it with prior approaches. Figure 1 depicts our architecture. It is designed to run on either a cluster of computing nodes or a multicore machine. Multiple solver instances (called *workers*) work on the same problem and share information through a central *broker*. The workers are SMT solvers instrumented to be able to export and import learned lemmas on the fly. Workers also track local statistics about lemma imports, exports, and filtering.

The central broker plays two roles. First, in the *control plane* (Fig. 1a), it manages the system by starting, configuring, monitoring, and terminating workers, and by monitoring the overall system and network health (through periodically transmitted ping/pong messages). Second, in the *data plane* (Fig. 1b), it controls system data flow by managing lemma exchange between workers and by tracking and monitoring solver and system-level lemma statistics. In particular, the data-plane broker (i) tracks which lemmas arrive from which individual workers and (ii) decides which lemmas to forward to which workers. This already enables a finer level of control than in previous approaches, where lemma sources are not tracked and static selection criteria are used to decide which lemmas to share. The broker tracks both control and data, including statistics such as the number of lemmas exported or imported so far, time spent in various phases of processing those lemmas, whether a worker has solved its copy of the problem, and so forth.

We advocate a simple hub-and-spoke architecture, similar to that used in SMTS [16]. Using a central broker simplifies coordination and does not require workers to synchronize with each other. We have also observed empirically that in our implementation, the broker is not a communication bottleneck (see Sec. VI). Our hub-and-spoke architecture tolerates worker failure and communication lag or failure. The design makes progress as long as the central broker and some workers are active. The broker is a single point of failure, but can be engineered to be robust.

A. Workers

As mentioned above, the workers are SMT solvers modified to support importing and exporting of learned lemmas during search. This allows for more fine-grained information sharing than prior approaches, where lemmas are only imported at decision level 0, and requires modifying the CDCL loop as shown in Algorithm 2. The loop now calls an export procedure whenever a new lemma is learned as a result of conflict analysis (Line 14). Additionally, during the propagation phase, the worker adds lemmas received from the broker to its database by invoking an import procedure (Line 7). While these changes to CDCL are non-trivial, we can often leverage extensions already present in CDCL SAT solvers to support SMT functionality such as theory propagation. These mechanisms can typically be repurposed for lemma sharing.

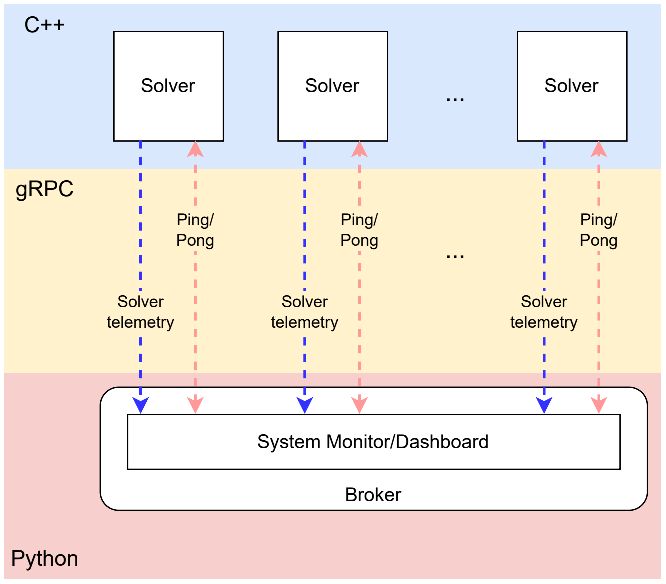
The worker sends telemetry to the broker whenever lemmas are exported or imported (Line 9 and Line 15). Each solver has a mechanism for locally filtering lemmas. The goal is to import and export only *useful* lemmas. We discuss various considerations for local filtering in Section V.

B. Central Broker

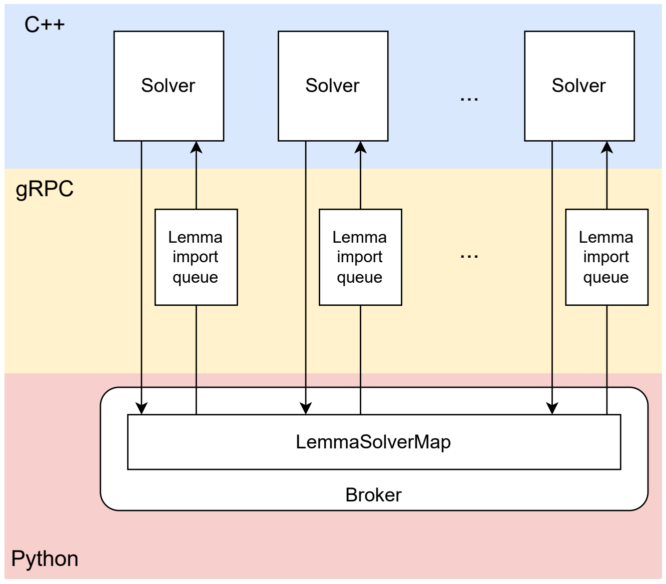
The central broker configures both the workers and network communication channels and manages both the control and data planes. During solving, it coordinates the exchange of information between workers and detects termination.

A major role of the central broker is to distribute lemmas learned by one worker to the other workers, while discarding duplicates and managing additional filters. Because multiple workers can learn and export identical lemmas, the broker ensures that each unique lemma is only forwarded (at most) once to each worker. Again, this offers a more fine-grained control mechanism than prior work, in which all lemmas up to a certain size are always shared (Z3) or lemmas are sampled randomly (SMTS) from the database of all shared lemmas.

The core broker algorithm is shown in Algorithm 3. The broker maintains two global variables: $archivedLemmas$ is the set of all lemmas it has received; and $lemmaSolverMap$ is a map from lemmas to worker ids that keeps track of the origin(s) of each lemma. When the broker receives a lemma, the lemma is canonicalized by sorting the set of its literals (Line 5). This ensures that one source of lemma redundancy is eliminated. The broker then uses this canonical form to detect whether the lemma is new (i.e., not in $archivedLemmas$) and to update $lemmaSolverMap$. Function *shouldSend* controls



(a) Control Plane



(b) Data Plane

Figure 1: Architecture of SMT-D

```
Algorithm 2: Modified CDCL(T) loop with sharing
Input  : an SMT formula $F$
Output: SAT or UNSAT
1  $clauseDB \leftarrow toCNF(F)$;
2  while $True$ do
3      do
4          $conflict \leftarrow BooleanPropagate(clauseDB)$;
5          $changed \leftarrow$ False;
6          if $conflict = \emptyset$ then
7              $newLemmas \leftarrow importLemmas()$;
8              $clauseDB \leftarrow clauseDB \cup newLemmas$;
9              $sendtelemetry()$;
10             $conflict, changed \leftarrow theoryCheck()$;
11     while $(newLemmas \neq \emptyset \vee changed) \wedge conflict = \emptyset$;
12     if $conflict \neq \emptyset$ then
13         $level, lemma \leftarrow resolveConflict(conflict)$;
14         $exportLemma(lemma)$;
15         $sendtelemetry()$;
16         $clauseDB \leftarrow clauseDB \cup lemma$;
17         if $level < 0$ then
18             return UNSAT;
19         backtrack($level$);
20     else
21         if $nextLiteral() = NULL$ then
22             return SAT;
```

```
Algorithm 3: The broker's core lemma exchange routine
1  $archivedLemmas \leftarrow \emptyset$;
2  $lemmaSolverMap \leftarrow \emptyset$;
3  while $True$ do
4      $\ell, w \leftarrow readMessage()$;
5      $\ell \leftarrow canonicalize(\ell)$;
6      if $\ell \in archivedLemmas$ then
7          continue;
8      $lemmaSolverMap[\ell].add(w)$;
9      if $shouldSend()$ then
10         for $\ell \in lemmaSolverMap$ do
11             $send(\ell, allWorkers \setminus lemmaSolverMap[\ell])$;
12             $lemmaSolverMap.pop(\ell)$;
13             $archivedLemmas.add(\ell)$;
```

the timing of when lemmas are transmitted to the workers. When *shouldSend* is true, the broker sends each lemma l stored in *lemmaSolverMap* to the workers that did not export it. We discuss implementation choices for *shouldSend* in Section V.

IV. Portfolio Strategies

Constructing effective strategies for portfolio solving with information sharing requires balancing trade-offs from a number of different goals:

- *Maximize diversity:* workers should work on different parts of the search space to avoid redundant work.
- *Share useful lemmas:* ideally, workers should export lemmas that are useful to all instances. A common heuristic for evaluating the value of a lemma is its size (i.e., number of literals in the clause). Smaller clauses are more likely to be useful, as they prune a larger portion of the search space.
- *Avoid overwhelming solvers:* each solver maintains a database containing both locally-learned lemmas and lemmas imported from the broker. Core solver perfor-

mance degrades as the size of the database grows. Sharing too many lemmas can thus be detrimental to overall system performance.

- *Manage communication overhead:* we do not want to overload the communication network with too much data, as this also slows down the system.

Our proposed architecture supports a wide variety of strategy options. We mention two general strategies here, and then discuss specific parameter settings used in our implementation in Section V. The first strategy is *delayed sharing*, which avoids sharing a large set of lemmas that all solvers discover locally. The second strategy is a novel approach to diversity that we call *guided randomization.*

A. Delayed Sharing

In initial experiments with an early prototype, we observed that for some large problems, workers initially export a large number of lemmas and delay calling the *importLemmas* procedure. Later, when they do try to import the lemmas, the system stalls due to the large amount of communication traffic. Telemetry revealed that this was caused by the initial preprocessing and theory reasoning performed by the solvers.

Before entering the CDCL loop proper, SMT solvers perform formula simplification, conversion to clausal form, and some eager theory reasoning. It is possible for solvers to produce many lemmas during this phase; if each worker is an instance of the same SMT solver, such lemmas are likely to be learned by all solvers working on the problem.To address this issue, we added a delayed sharing mechanism, which ensures that only lemmas learned *after* the preprocessing phase are exported. Enabling this mechanism boosts performance on all of our benchmarks.

B. Guided Randomization

Baseline mechanisms for diversifying solver behavior include selecting different random seeds and modifying solver configurations to ensure that different instances use different search parameters. However, these basic mechanisms have diminishing benefit as we increase portfolio size, as we show in Section VI-B. Using the telemetry collected by the broker, we can observe the number of uniquely learned lemmas (i.e., those learned by a single worker). This metric is a reasonable proxy for system diversity, and indeed, in early experiments, we observed that this number plateaus as we scale the number of workers.

We address this problem by dividing the pool of workers into two clusters, a standard cluster and a *noisy* cluster. Each cluster uses different levels of randomness and different scoring and filtering heuristics. Scoring and filtering can also treat lemmas local to the cluster differently than clauses from other clusters. The noisy cluster uses a high degree of randomness. Intuitively, we expect that solvers in this cluster will learn mostly useless clauses, because they are using heuristics that are far away from the default configurations which have been tuned to be effective. They are also likely to end up exploring parts of the search space that low-randomness solvers ignore. But once in a while, noisy solvers may get lucky and learn clauses that can be useful to solvers in the other cluster.

To maintain diversity in the noisy cluster, we keep the clause databases for solvers in the cluster somewhat isolated. We do this by configuring solvers in noisy clusters to ignore each other and only import lemmas that the central manager determines are highly likely to be useful, (*e.g.,* unit clauses). We discuss a concrete instantiation of this strategy in the next section.

V. Implementation

SMT-D is a distributed SMT solver that implements our proposed architecture and strategies.For the worker instances, we use a version of cvc5 with the main loop modified to support importing and exporting clauses, as discussed in Section III-A. Workers run in separate processes, and each worker process has a separate *wrapper* thread that manages the control plane interface and networking details.

The central broker is written in Python. Communication between broker and workers is implemented with gRPC [9]. We chose gRPC instead of lower-level mechanisms like sockets, because gRPC's high-level API provides better monitoring capabilities and has sufficient performance for (at least) 64 solvers. gRPC also allows us to abstract the parallel and distributed aspects of the system. Thus, SMT-D can be deployed either on a single multicore machine or on a cluster of machines in the cloud.

To export lemmas, we serialize them as strings in the SMT-LIB format [4]. Correspondingly, lemma import requires parsing SMT-LIB strings. This adds some overhead[1] but provides a significant interoperability advantage, as all SMT solvers can parse and print terms in SMT-LIB format. More compact formats could be used at the cost of increased implementation effort and reduced interoperability. For example, SMTS uses a dedicated binary format, but this limits the choice of solvers to those that support this format. Choosing SMT-LIB reduces the cost of adding additional solvers beyond cvc5 to SMT-D.

As explained previously, SMT-D implements comprehensive telemetry for both the control and data planes. We found this real-time information about the solving process at both the local and global levels to be crucial when debugging the system, evaluating different portfolio configurations, and evaluating lemma scoring and filtering strategies. The implementation is heavily parameterized, so that whenever possible, users can choose configuration options at runtime, rather than having to change hard-coded configuration settings.

A. Local Filtering

Several considerations must be taken into account at the worker level. SMT solvers can dynamically create new atoms and new symbols during search. This poses a soundness problem in a distributed setting as one must ensure that new symbols created by a solver instance are interpreted

[1]So far, this has not been a performance limiter, as analysis shows that individual cvc5 workers can import at least 1,000 lemmas/second with <5% parsing overhead. None of the benchmarks reach that level.

consistently by other instances. We currently avoid this issue at the export stage by filtering out lemmas that contain symbols not present in the original formula.[2] New theory atoms are fine as long as they do not introduce new symbols. More sophisticated approaches are possible, but require a mechanism for exporting the definitions of new symbols in a canonical way. Implementing such a mechanism requires extending the baseline SMT solver in a non-trivial way, and we leave it for future work.

As mentioned, our primary goal when filtering is to only export useful lemmas. As in prior work, we use the number of literals in the lemma as our main export filter.

Importing lemmas has a cost. The central broker aims to limit redundancy by only sending a given lemma once to each worker. It is still possible for a worker to produce a lemma internally before learning that another worker has produced the same lemma. Thus, we check in the import procedure whether an imported lemma has already been discovered locally. If so, we drop it. This can be implemented efficiently using mechanisms such as hashing and Bloom filters.

B. Sending Lemmas from the Broker

Our broker uses two indicators to determine when to send lemmas. The first is the wall clock time elapsed since the last lemma transmission. The other is the number of unsent lemmas for a particular worker in the $lemmaSolverMap$ map. Function $shouldSend$ returns true if the elapsed time is greater than a parameter *delay* or if the number of unsent lemmas is larger than a threshold *maxQueueSize*. By setting these two parameters, the broker can implement different communication policies. It can send lemmas in size-driven batches (like SMTS [15]), in time-driven epochs (like Mallob [19]), or both. We found empirically that so far, the best results come from sharing lemmas individually as soon as they are received. The current sharing limiter is cvc5 parser performance, which supports importing at least 1,000 lemmas per worker per second. With clause sharing filtered by size ≤ 8, only one of the benchmarks approaches that limit, even with 64 workers. If we encounter network bandwidth limitations at some point, we expect that time-driven epochs will provide the best efficiency.

C. Monitoring

SMT-D monitors the number of lemmas imported and exported by each worker. Information from solver wrappers is used to monitor message latency and broker/solver roundtrip times. The $lemmaSolverMap$ map also tracks how many solvers independently learned each lemma, that is, the number of lemmas learned by exactly one solver, two solvers, and so forth. This helps dynamically measure diversity in the system, including the amount of redundant work being performed by different solvers. The broker also maintains its own counts of the number of exported and imported lemmas for each worker. Mismatches between the numbers stored in the broker and the numbers reported by the workers mean that the system is overloaded (thus messages are late or dropped) or that there is a bug. During the development of SMT-D, the monitor helped detect multiple bugs and helped inform the design of our lemma-sharing heuristics.

[2]This problem does not occur in the problems in our evaluation, as problems in these logics do not introduce new symbols.

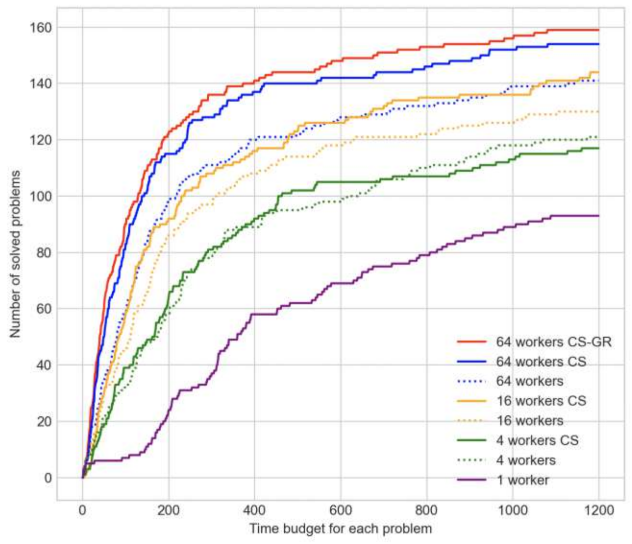


Figure 2: Scalability of SMT-D.

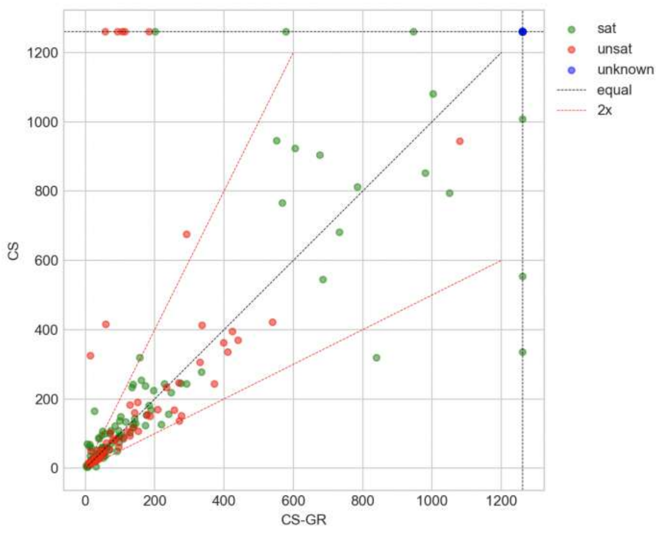


Figure 3: Guided Randomization (CS-GR) vs naive Clause Sharing (CS). Dots on the upper and right-most edges are problems that time out with CS and CS-GR, respectively.

VI. Evaluation

We measure SMT-D performance on the set of benchmarks used in [22], which consists of 214 challenging benchmarks taken from the Cloud track of SMTCOMP22 [18] and other problems from the SMT-LIB benchmark library [3]. The benchmarks come from five SMT-LIB logics: QF_LRA (139), QF_IDL(48), QF_LIA (16), QF_UF (7), and QF_RDL (4).

Benchmarks		SMT-D baseline		SMT-D 64x CS		SMT-D 64x CS-GR		SMTS baseline		SMTS 64x CS		CVC5-P 64x	
Category	Count	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2
QF_LRA	139	90	154	121	61 (↓60%)	120	60 (↓**61%**)	117	69	**127**	**41** (↓41%)	99	130 (↓16%)
QF_IDL	48	1	114	20	72 (↓37%)	**21**	**70** (↓**39%**)	8	99	15	82 (↓17%)	5	107 (↓6%)
QF_LIA	16	0	38	8	22 (↓42%)	9	20 (↓**47%**)	11	13	**14**	**11** (↓15%)	1	36 (↓5%)
QF_UF	7	2	14	3	11 (↓21%)	**7**	**2** (↓**86%**)	6	5	6	3 (↓40%)	4	9 (↓36%)
QF_RDL	4	0	10	2	6 (↓40%)	**2**	**6** (↓**40%**)	0	10	0	10 (0%)	0	10 (0%)
SAT	115	52	172	86	83 (↓52%)	86	82 (↓**52%**)	83	87	**99**	**44** (↓49%)	59	151 (↓12%)
UNSAT	85	41	124	68	55 (↓56%)	**73**	**43** (↓**65%**)	59	75	63	63 (↓16%)	50	106 (↓15%)
UNKNOWN	14	0	34	0	34 (0%)	0	34 (0%)	0	34	0	34 (0%)	0	34 (0%)
ALL	214	93	330	154	171 (↓48%)	159	159 (↓**52%**)	142	196	**162**	**141** (↓28%)	109	291 (↓12%)

Table I: Results comparing SMT-D with other distributed solving tools. PAR-2 scores in thousands.

The goal of our evaluation is to understand the value and potential of our clause-sharing mechanism. Our first set of experiments evaluates different options and configurations of SMT-D (see Section VI-B). This experiment shows the effectiveness of clause sharing over no sharing and the value of guided randomization. Our second set of experiments compares SMT-D with other tools.

A. *Configuration*

We use a competition build of cvc5 with the elective CLN and GLPK build options enabled. For each logic, we configure cvc5 workers with different sets of options to enhance diversity. These option sets are listed in Table II and are based on the authors' knowledge of the tool and the configurations typically used in the SMT competition. We populate the portfolio by first instantiating a cvc5 instance for each set of options. If we have more workers available in the portfolio, we cycle through the different option sets again, but this time using a different decision engine from the default one for that logic (--decision=justification if the default is --decision=internal and --decision=internal otherwise). After this, we continue to cycle through the different sets of options, this time using only --decision=internal and using a different random seed for each instance. When using noisy solvers, only solvers with --decision=internal are used for the noisy partition.

Table II lists the different sets of options used for each logic. The first set of options listed for each logic is the one used when running a single instance of cvc5 for that logic.

In all experiments, we set the timeout for solving each query to be 1200 seconds, the same timeout used in the parallel and cloud tracks of the SMT competition (in both 2022 and 2023) [18]. Experiments were performed on Amazon EC2 c6a.48xlarge instances, with 96 physical cores and 384 GB of RAM.

Our main metric used for comparison is the *PAR-2 score* used in [22] and the annual SAT competition. PAR-2 is the sum of run times for all instances, but where unsolved instances receive a score of twice the timeout value ($1200 \times 2 = 2400$). This provides a single metric that takes into account both runtime and number of benchmarks solved. The lower the PAR-2 score, the better. We also use *cactus plots* to show the number of solved instances (y-axis) within a limit of s seconds per instance (x-axis). We are primarily interested

Logic	**Options**
QF_LRA, QF_RDL (option set 1)	--miplib-trick true --miplib-trick-subs 4 --use-approx true --lemmas-on-replay-failure true --replay-early-close-depth 4 --replay-lemma-reject-cut 128 --replay-reject-cut 512 --unconstrained-simp true --use-soi true
(option set 2)	--restrict-pivots false --use-soi true --new-prop true --unconstrained-simp true
(option set 3)	(defaults only)
QF_LIA, QF_IDL (option set 1)	--miplib-trick true --miplib-trick-subs 4 --use-approx true --lemmas-on-replay-failure true --replay-early-close-depth 4 --replay-lemma-reject-cut 128 --replay-reject-cut 512 --unconstrained-simp true --pb-rewrites true --ite-simp true --simp-ite-compress true --use-soi false
(option set 2)	--miplib-trick true --miplib-trick-subs 16 --use-approx true --lemmas-on-replay-failure true --replay-early-close-depth 4 --replay-lemma-reject-cut 16 --replay-reject-cut 64 --unconstrained-simp true --pb-rewrites true --ite-simp true --simp-ite-compress true --use-soi true
(option set 3)	(defaults only)
QF_UF	(defaults only)

Table II: Options used in cvc5 portfolios

in the effectiveness of different parallelization strategies and implementations.

B. *Scalability and Effectiveness of Guided Randomization*

We first report on scalability experiments of SMT-D, both with and without sharing. We also show the effect of adding guided randomization. When using guided randomization, we divide the portfolio into two clusters: a *standard* cluster, which uses default cvc5 randomness settings, and a *noisy* cluster, which assigns the cvc5 `rnd_freq` option to 75%. This option controls how often the SAT decision tries to pick a random variable instead of a heuristically-driven choice. We assign 25% of the workers to the noisy cluster and 75% to the standard cluster.[3] Solvers in the standard cluster import and export clauses of length ≤ 8. In the noisy cluster, clauses of length ≤ 4 are exported, but only unit clauses are imported.

To distinguish the different configurations of SMT-D, we use *CS* for configurations with clause sharing and *CS-GR* for configurations with clause sharing and guided randomization. Fig. 2 shows how different configurations of SMT-D scale with the number of workers. The figure includes results for baseline cvc5, portfolios of 4, 16, and 64 workers, with and without sharing, and a run of 64 workers with guided randomization. Specific numbers for three of the configurations (baseline, 64x CS, and 64x CS-GR) can be found in Table I.

We observe that SMT-D scales nicely when going from 1 to 64 solvers. In addition, clause sharing improves performance for all portfolio sizes greater than four, and guided randomization provides an additional boost. Our experiments showed that guided randomization does not help much until we reach portfolio sizes of more than 32. We suspect additional portfolio members add diversity until a point of diminishing returns where the guided randomization helps. This is why we only include results for CS-GR for a portfolio of 64. A comparison of the 64x CS configuration with and without guided randomization is shown in Fig. 3. While there is orthogonality, overall CS-GR improves performance, including by more than 2x for a significant number of problems (dots to the left of the top "2x" line). As a whole, among all instances solved by both CS and CS-GR, there are 24 instances where CS-GR is more than 2x faster than CS, and only 5 instances where CS-GR is 2x slower. CS-GR solves 5 more problems, and improves the PAR-2 score by 12k (7%) over CS.

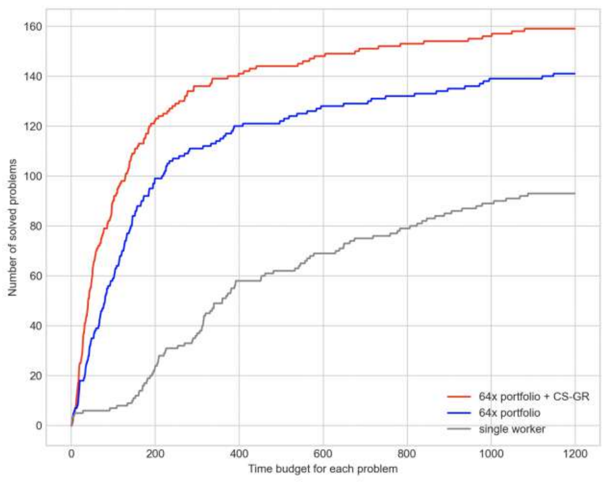
Though we did not measure it precisely, total memory consumption per solver is relatively stable, which is good news for the distributed case where cores do not share memory. Broker memory was not a significant issue in these experiments. Detailed studies of memory usage will be an important part of ongoing development of the tool.

C. Comparison with State-Of-The-Art Tools

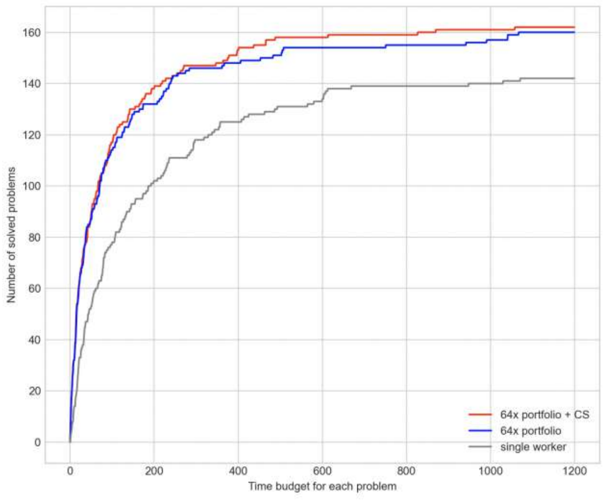
We next compare SMT-D with SMTS [16],[4] the strongest solver in quantifier-free divisions of SMTCOMP22's cloud track,[5] and CVC5-P, the partitioning solver from [22].

We use SMTS with sharing on and partitioning off. The reason for not enabling the partitioning capability is simple and deliberate: our goal is to understand and compare only the clause-sharing capabilities of the two frameworks. Results of SMTS with both sharing and partitioning enabled would be inconclusive, as it would be difficult to figure out which technique contributed what. And it would not be an apples-to-apples comparison, as our approach does not yet integrate partitioning. We anticipate integrating clause-sharing with partitioning in future work. In contrast, in our comparison with CVC5-P, we do use the partitioning capabilities of CVC5-P. But this is again deliberate as our goal with that comparison is different, namely to explore how clause sharing compares to partitioning when using the *same* underlying solver (SMTS uses a *different* underlying solver, namely OPENSMT2).

a) Comparison to SMTS*:* It is important to note that on this benchmark set, OPENSMT2, the baseline solver for



(a) SMT-D



(b) SMTS

Figure 4: Comparing SMT-D's and SMTS' improvement over a single base solver.

[3]These percentages were chosen based on an empirical analysis of a small sample of possible values. We plan to do a more extensive evaluation of these parameters in the future.

[4]We used commit 29d51340 from the cube-and-conquer branch, as recommended to us by the SMTS authors.

[5]SMT-COMP 2023's cloud track omitted all quantifer-free divisions.

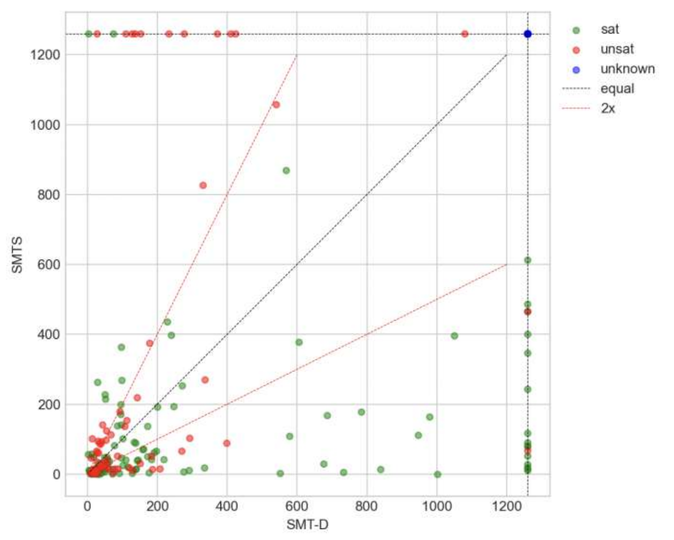


Figure 5: SMT-D 64x CS-GR vs SMTS 64x. Dots on the upper and right-most edges are problems that time out for SMTS and SMT-D, respectively.

SMTS, is stronger than cvc5.[6] However, the best configuration of SMT-D (64 CS-GR) improves this situation significantly, as can be seen by the relatively larger gap between the best configuration and the "single worker" configuration in Fig. 4. Table I shows that overall, in terms of benchmarks solved, the best configuration of SMT-D (64 CS-GR) solves almost the same number of problems as the best configuration of SMTS, despite the large difference in their base solvers. Compared to the baseline, the best configuration of SMT-D improves the overall PAR-2 score by 52% (for SMTS, this number is 28%) and solves 66 more problems (compared to 20 more problems solved by SMTS). Moreover, for the 48 QF_IDL benchmarks and for the UNSAT benchmarks as a whole, cvc5 goes from performing worse than SMTS when comparing baselines to performing better when comparing the best version of each.

Although one might hope for even better scaling as the level of parallelism increases, it is important to keep in mind that SMT is a hard problem and is not easily parallelizable. Thus, we don't expect to be able to achieve linear speed-up. Rather, we hope to solve problems beyond the scope of standalone solvers, and indeed, we see that this is the case. In many applications, the number of problems solved in a fixed time matters. We can also see (Figure 5) that SMTS and SMT-D solve a different subset of the benchmarks, so we know further improvement is still possible.

b) Comparison to partitioning cvc5: CVC5-P, the state-of-the-art parallel/distributed implementation of cvc5, uses a combination of portfolios and partitioning strategies. We implemented and ran the hybrid multijob approach of [22] and compared it with SMT-D. Fig. 6 and Table I show that SMT-D is significantly more effective at utilizing 64 copies of cvc5, resulting in a 52% improvement in PAR-2 score (vs 12% improvement by CVC5-P), and in 50 more problems being solved (159 vs 109).

[6]One reason for this is that the benchmarks we are using, from [22], were selected specifically because they are challenging for cvc5.

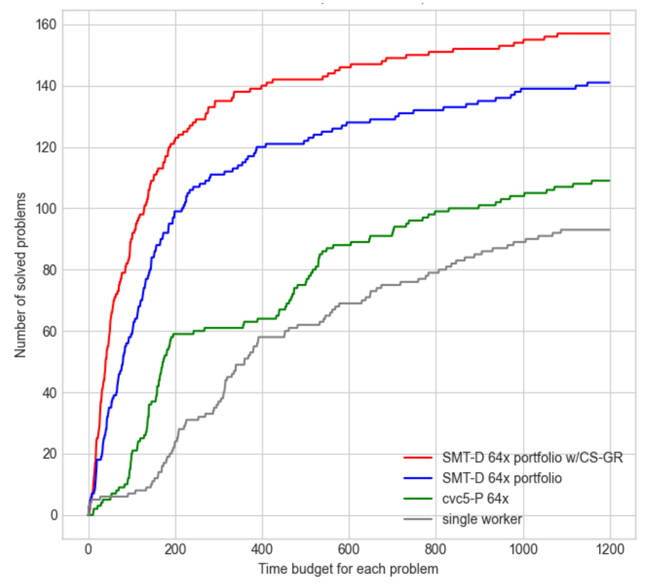


Figure 6: SMT-D and CVC5-P, 64 workers vs 1 worker.

	single worker	8x portfolio + CS
Z3	190	205
SMT-D	219	174

Table III: Comparison between SMT-D and Z3 on 129 benchmarks. Entries show PAR-2 scores in thousands.

D. *Comparison to a Legacy Version of* Z3

Z3 was the first SMT solver to implement a portfolio approach with clause sharing. However, this functionality is no longer supported in modern versions of Z3, and the latest release that we could find with this functionality is version 2.15 (Windows-only, from 2009). We attempted a comparison, for completeness, but are only able to draw limited conclusions, for various reasons, including: (i) Z3 2.15 runs on a different operating system than our other solvers; (ii) it crashes on any configuration with more than eight solvers; and (iii) it fails (parsing or execution) on 85 problems in our modern set of 214 benchmarks. When run on the remaining 129 SMT benchmarks, we obtain the results shown in Table III. However, even these results must be taken with a grain of salt, as they show that Z3 performs *worse* when enabling clause sharing, perhaps because of instability of the 2.15 implementation on modern benchmarks. Thus, while this early work in Z3 was important pioneering work, we believe that a fair comparison can only be achieved if the sharing functionality is restored in a modern version of Z3.

VII. Conclusion

SMT-D is a promising advancement in the realm of parallel, portfolio-based SMT solving. Leveraging a hub-and-spoke architecture with a tight CDCL(T) integration, lemma sharing, and guided randomization, SMT-D demonstrates significant improvements in scalability, outperforming not just sequential cvc5, but also pure portfolio (with sharing), and CVC5-P (portfolio with partitioning). In addition, SMT-D demonstrates more improvement from clause sharing than SMTS and an early version of Z3 and has performance that is overall comparable with and complementary to the state of the art.

While SMT-D demonstrates solid progress in distributed SMT solving, many opportunities for future work remain. These include further parameter tuning, deeper integration with the underlying SAT solver, handling internally-introduced symbols, exploring additional sources of diversity (including using different solvers in the portfolio, such as OpenSMT2 or Z3), exploring additional filtering and redundancy-detection heuristics, and combining our approach with partitioning-based parallelism. In addition, we plan to extend the implementation and evaluation of SMT-D to the full set of logics and benchmarks in SMT-LIB.

References

[1] Tomás Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. *CoRR*, abs/1505.03340, 2015.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *TACAS '22*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability, Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 33, pages 825–885. IOS Press, February 2021.

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999. Springer Berlin Heidelberg.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. Springer Berlin Heidelberg.

[8] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[9] Google. grpc.io. https://grpc.io/. [Accessed 15-Mar-2023].

[10] A. Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*, 2015.

[11] Youssef Hamadi and Lakhdar Sais, editors. *Handbook of Parallel Constraint Reasoning*. Springer, 2018.

[12] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, 2016.

[13] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'11, page 50–65, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving SAT in grids. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 430–435, 2006. Springer Berlin Heidelberg.

[15] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. Clause sharing and partitioning for cloud-based SMT solving. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis*, pages 428–443, Cham, 2016. Springer International Publishing.

[16] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. SMTS: distributed, visualized constraint solving. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 534–542. EasyChair, 2018.

[17] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, nov 2006.

[18] SMT-COMP Organizers. SMT-COMP. https://smt-comp.github.io/, 2023.

[19] Dominik Schreiber and Peter Sanders. Scalable SAT solving in the cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing.

[20] Roberto Sebastiani and Silvia Tomasi. Optimization modulo theories with linear rational costs. *ACM Transactions on Computational Logic*, 16, 10 2014.

[21] Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is at least 14. In *International Conference on Theory and Applications of Satisfiability Testing*, 2022.

[22] Amalee Wilson, Andres Nötzli, Andrew Reynolds, Byron Cook, Cesare Tinelli, and Clark W. Barrett. Partitioning strategies for distributed SMT solving. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 199–208. IEEE, 2023.

[23] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. A concurrent portfolio approach to SMT solving. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 715–720, 2009. Springer Berlin Heidelberg.

[24] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.

Formal Methods in Computer-Aided Design 2024

Modernizing SMT-Based Type Error Localization

Max Kopinsky
McGill University
Montréal, Quebec
max.kopinsky@mail.mcgill.ca

Brigitte Pientka
McGill University
Montréal, Quebec
bpientka@cs.mcgill.ca

Xujie Si
University of Toronto
Toronto, Ontario
CIFAR AI Research Chair
six@cs.toronto.edu

***Abstract*—Traditional implementations of strongly-typed functional programming languages often miss the root cause of type errors. As a consequence, type error messages are often misleading and confusing - particularly for students learning such a language. We describe Tyro, a type error localization tool which determines the optimal source of an error for ill-typed programs following fundamental ideas by Pavlinovic et al. : we first translate typing constraints into SMT (Satisfiability Modulo Theories) using an intermediate representation which is more readable than the actual SMT encoding; during this phase we apply a new encoding for polymorphic types. Second, we translate our intermediate representation into an actual SMT encoding and take advantage of recent advancements in off-the-shelf SMT solvers to effectively find optimal error sources for ill-typed programs. Our design maintains the separation of heuristic and search also present in prior and similar work. In addition, our architecture design increases modularity, re-usability, and trust in the overall architecture using an intermediate representation to facilitate the safe generation of the SMT encoding. We believe this design principle will apply to many other tools that leverage SMT solvers.**

Our experimental evaluation reinforces that the SMT approach finds accurate error sources using both expert-labeled programs and an automated method for larger-scale analysis. Compared to prior work, Tyro lays the basis for large-scale evaluation of error localization techniques, which can be integrated into programming environments and enable us to understand the impact of precise error messages for students in practice.

I. Introduction

Many strongly typed programming languages, such as OCaml [1], allow programmers to omit type annotations from their code; despite these omissions, *type inference* automatically reconstructs the types of all expressions in the program based on the contexts in which they appear. For well-typed programs, type inference saves the programmer much time and effort. However, for ill-typed programs, the situation can be exactly the opposite [2]. Type errors are discovered when the compiler finds inconsistencies during type inference, but figuring out *root causes* is much harder. The location where compiler fails is usually *not* the place to fix the reported type errors. As a result, type errors are often misleading or confusing. Such errors increase debugging time for programmers. In the case of novices, such errors discourage them from learning the language at all [3]. Even tools designed to assist novices, such as Helium [4], frequently produce such misleading errors.

The importance, and difficulty, of finding accurate causes of type errors ("localization") has a long-studied history. A system for recording "reasons" that may explain type mismatches was implemented in Wand's SPS [5] in 1986 [6]. Improvements to Wand's method include the recent HM$^\ell$, which turns the problem of explaining the "reasons" into a data flow problem [7]. Other recent approaches use machine learning techniques to localize errors [8], [9] but without any formal guarantees.

There is also a class of techniques based on heuristic search. Type inference is naturally expressed as a constraint-solving problem [10], [11], [12], even for more complex type systems, e.g. [13]. By heuristically attributing weights to each constraint, techniques for constrained optimization can be applied. Such techniques can involve custom frameworks and solvers, as in Mycroft [14]; or more generalized tools such as SMT solvers.

Our work builds on prior work using SMT solvers. Cutting-edge SMT solvers, such as Z3 [15], are being actively developed and steadily improved. These improvements cut down on memory usage and runtime, enabling SMT solvers to handle increasingly large problem instances. Localization approaches that leverage such tools therefore benefit from continuous improvements to SMT solvers.

Pavlinovic et al. developed MinErrLoc [16], the state-of-the-art type error localization tool based on a variant of SMT called MaxSMT. In the case of an ill-typed program, there is no satisfying assignment for the typechecking constraint problem. from type inference. Instead, MinErrLoc seeks a minimum-weight set of constraints explaining why no solution exists. Although effective at the time of its publication, MinErrLoc depends on a customized version of CVC4 [17], rather than off-the-shelf MaxSMT solvers, and was not maintained after its original publication in 2014. Thus, MinErrLoc suffers from package rot and requires significant effort to run. Our objectives were to bring the MinErrLoc approach up to modern standards, and make it possible to leverage modern off-the-shelf MaxSMT solvers as originally intended.

Our main contribution is a new type error localization tool, Tyro,[1] inspired by the fundamental work of Pavlinovic et al. Tyro incorporates a new encoding for constraints resulting from polymorphic types, and is implemented with a two-stage design. The first stage generates a human readable intermediate representation of the typechecking constraint problem. Separate aspects of the problem are kept apart, increasing readability. The second stage processes the intermediate representation into an SMT-LIB encoding [18], bringing together separate

[1]https://github.com/JKTKops/tyro

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_11

aspects of the problem to form the encoded constraint system. We also found that this architecture made the individual stages easier to debug, and therefore increases trust in the overall system. Though the intermediate representation is specific to our system, we anticipate that the same ideas could be applied to a wide range of systems that leverage SMT.

Our experimental evaluation expands the evaluation of the MinErrLoc approach to a much larger dataset, Validating the accuracy of Pavlinovic et al.'s approach, but also highlights the need for better heuristics on some classes of programs. We performed accuracy evaluations with a small expert-labeled dataset, and both accuracy and performance evaluations with a large dataset automatically extracted from student code in a large, introductory OCaml course.

II. Overview of the MinErrLoc Approach

Since our work builds on MinErrLoc [16], a brief overview of its key ideas is warranted.[2] Primarily, we review the localization problem, the meaning of Pavlinovic et al.'s "minimum error source" heuristic, and its reduction to MaxSMT.

Type errors result from (often minor) mistakes on the part of a programmer. Correcting these mistakes will resolve the type error. The program region containing the mistakes(s) is called the "root cause" of the type error.

The localization problem that we aim to solve is, given a program P that exhibits a type error, to identify the root cause. This is an inherently ambiguous problem, because we cannot be certain exactly what the programmer intended. The MinErrLoc approach follows Occam's Razor – the simplest explanation is probably the correct one.

A. *Minimum Error Sources*

An "error source" is a set of program locations which resolve the type error if removed from the program.[3] The root cause of the type error must be at a subset of an error source.

Not all error sources are equally likely to contain the true root cause, however. The MinErrLoc framework provides an opportunity to specify a weight for every program location. A "minimum error source" is an error source whose total weight is minimum. The framework allows these weights to be assigned independently of constraint generation. Locations can also be set as "hard constraints" to tell the solver that they should not be considered in potential error sources.

Consider this recursive OCaml program for finding the length of a list, which contains a bug:

```
let rec len = function
  | [] -> 0.
  | _ :: xs -> 1 + len xs
```

This program is ill-typed, because the first arm produces a `float`, but the second arm's use of `+` means it produces an `int`. There is more than one way to explain this error. One possible error source is `0.`, the float-valued first arm. Replacing this with an int-valued expression would resolve the error. Another possible error source is the use of `+`. Replacing `+` with a `float`-valued function could also resolve the error.

If we use a trivial weighting heuristic, which simply assigns a weight of 1 to every location, then both error sources will be minimal. However, domain knowledge might suggest that `0.` is far more likely to be the true error source. A weighting heuristic which considers the complexity of a program location, or which penalizes function calls, might result in `0.` as the unique minimum error source.

The MinErrLoc framework ensures that the constraint generation algorithm is independent of weight assignments. This allows the framework to be re-used with different weighting heuristics.

B. *Reduction to MaxSMT*

MaxSMT is a variation of the SMT problem. Recall that SMT may be defined as the decision problem asking whether a set $\mathcal{C}$ of propositional clauses is satisfiable. MaxSMT instead seeks a maximum subset $\mathcal{C}' \subseteq \mathcal{C}$ such that $\mathcal{C}'$ is satisfiable. Note that maximizing the size of $\mathcal{C}'$ corresponds to minimizing the size of $\mathcal{C} \setminus \mathcal{C}'$, which will correspond to an error source. We may take this generalization of SMT two steps further. First, we may include a *weighting heuristic*, a function $w : \mathcal{C} \rightarrow \mathbb{N}$. Rather than seeking a subset $\mathcal{C}'$ of maximum size, we seek a subset which maximizes $\sum_{c \in \mathcal{C}'} w(c)$. This corresponds to the weighting heuristic for program locations mentioned above. Finally, we may allow some clauses to be "hard constraints," which *must* be satisfied by the assignment. The resulting problem is known as *Partial Weighted MaxSMT*, but we will call it MaxSMT for brevity.

It would be easy to translate the typing constraints directly into MaxSMT constraints. Constraints in OCaml programs are equality constraints between types. Equalities between (mono)types, and the types themselves, can be encoded using the Theory of Inductive Datatypes [19], which has been added to the SMT standard and is supported by SMT solvers such as Z3 [15], [18]. A datatype ("sort") is created in SMT which represents OCaml types. The OCaml types are then encoded as values of this SMT datatype.

However, this encoding would not produce error sources - it would produce sets of typing constraints. If several constraints arise from the same program point, the solver would be allowed to independently decide whether or not to satisfy them. Instead, we must force the solver to decide on a location-by-location basis. This is further complicated by the fact that the locations are not disjoint – The location corresponding to an expression contains all of the locations corresponding to its subexpressions. This tree structure is known as the *abstract syntax tree*, or AST, of the program.

To accomplish this, weights are associated with program locations, rather than constraints. The encoding of the constraints into MaxSMT incorporates information about the shape of the AST. The encoding of the shape is such that removing a location also implicitly removes all of its children. Otherwise we could be left with constraints to satisfy which are no longer

[2]Overviews of OCaml's polymorphic types and of classical type inference for the system can be found in the Appendix.

[3]"Remove" here means to replace by `failwith "removed"`. Literally deleting the location would almost always result in syntax errors.

in the program. Our variation of the encoding is discussed in detail below.

III. Tyro Architecture

Tyro uses a modular, two-stage software architecture. The stages are implemented as separate "frontend" and "encoder" tools. The input to the frontend is an OCaml program, and the output is an Intermediate Representation of the constraints. The encoder accepts this IR, and outputs an SMT-LIB script, which is then be passed to an off-the-shelf MaxSMT solver.

A. Frontend

The frontend's job is to extract a set of typing constraints from an OCaml program. We implemented it by modifying EasyOCaml [20]. EasyOCaml is a tool with improved error message quality for OCaml, and has also been modified for constraint generation in other work [16].

First, a set of constraints are generated, including our representation of polymorphic types. Then, the collected constraints are encoded into the intermediate representation.

The constraint generation is a modification of existing constraint-generation approaches [10], [13], [16]. As a reminder, we focus on an idealized fragment of OCaml, shown in Figure 2.

The fragment supports variables, lambda abstraction, function application, conditionals, and local variable bindings. The types g are the "ground types", such as `int`, `float`, or `string`. Types α represent globally unique type variables. These variables are *monomorphic* - they represent a single as-yet unknown type. Polytypes, on the other hand, may universally quantify some or all of the variables in a monotype, resulting in a template that can be re-used with multiple different types.

B. Polymorphic Types

Polymorphic types are a fundamental challenge for constraint-based type inference [10], [12]. When inferring a type for a polymorphic binding, a set of constraints will be generated. Some of these constraints will refer to the polymorphic variables in the type of the binding. Whenever the binding is used, copies of these variables are created in a process called *instantiation*. Every copy of these variables must be independent from the others. But every copy is also subject to the same constraints as the original. The solution taken by MinErrLoc is to also copy all of the constraints. Our approach instead encodes these constraints as abstractions, allowing the MaxSMT solver decide when, or indeed if, the copies should be created.

Since constraints associated with polytypes need to be recorded, a constraint set is attached to every polytype. "Type schemes" are a common approach to this in constraint-based systems [11], [13], [16]. After inferring the type for a binding **`let`** $x = e_1$ **`in`** e_2, the variable x will be added to the typing environment. Its type will have the form:

$$\forall \vec{\alpha}.(\mathcal{C}_x \Rightarrow \alpha_x)$$

where $\mathcal{C}$ is the associated set of constraints, and α_x is the type variable created for e_1. We write simply $x : \alpha_x$ if $\vec{\alpha}$ and $\mathcal{C}_x$ are both empty.

When x is later used, rather than create copies of the constraints in $\mathcal{C}_x$, we emit an "instantiation constraint." These constraints are of the form $x(\bar{\beta})$, and have appeared previously in other Hindley-Milner-style systems [11]. The constraint $x(\bar{\beta})$ represents the entire constraint set $\mathcal{C}_x[\vec{\beta}/\vec{\alpha}]$. That is, the capture-avoiding substitution of the variables $\vec{\beta}$ for the variables $\vec{\alpha}$ in a copy of C_x. Since instantiation constraints represent a set of regular typing constraints, they can appear wherever a set of typing constraints can appear.

C. Constraint Generation

A typing constraint in Tyro takes the form $\tau_1 =^{\ell} \tau_2$. This is a simple equality between two types, annotated with the program location ℓ where it was created. Since we need these locations to create the constraints, we ensure that the AST nodes are annotated with locations as well.

Unlike MinErrLoc, our frontend does not encode the structure of the AST into the typing constraints. To improve modularity and reusability, and to facilitate debugging, we keep this information separate for as long as possible. This, along with instantiation constraints, simplifies the typing rules significantly. The rules are formulated with a similar constraint typing relation, of the form:

$$\mathcal{C}; \Gamma \vdash e : \alpha$$

$\mathcal{C}$ is the set of constraints which have been emitted by inference for e. Γ is the typing environment in which inference for e should occur; Γ maps variable names to type schemes. e is a program expression, and α is its inferred type.

Note that the relation always relates an expression to a type variable. This means that we cannot infer the type `int` for the expression `0` - we must instead assign a new type variable α_0 and emit a constraint $\alpha_0 =$ `int`. This prevents a loss of information. If we could infer the type `int` directly, and the expression `0` were the root cause of the type error, there would be no link back to this source location in the constraint set [16]. The typing rules are shown in Figure 1.

Look in particular at the rules Var and Let, which are the main distinction from other constraint-based systems. In the case of variables, we look up the type scheme from the environment. Then we create new type variables to instantiate all variables in $\vec{\alpha}$. However, we do not then copy $\mathcal{C}_x$. Instead, we emit an instantiation constraint (with a location annotation). For let bindings, the difference is similar. Systems such as MinErrLoc emit the entire constraint set $\mathcal{C}_1[\vec{\beta}/\vec{\alpha}]$ where we emit the instantiation constraint $x(\vec{\beta})$. This instantiation constraint is necessary to ensure the consistency of $\mathcal{C}_1$ - otherwise, if all uses of x were removed from the program, all constraints in $\mathcal{C}_1$ would be lost [10], [16].

The constraint generator is implemented as a modification of EasyOCaml [20]. EasyOCaml is implemented as a fork of `ocamlc`, the OCaml compiler. This unfortunately pins it to a particular version of OCaml, which is not recent. In order to

$$\frac{\alpha \text{ new}}{\{\alpha =^{\ell} \textbf{int}\}; \Gamma \vdash n^{\ell} : \alpha}\ \textsc{Int} \qquad \frac{\alpha \text{ new}}{\{\alpha =^{\ell} \textbf{bool}\}; \Gamma \vdash b^{\ell} : \alpha}\ \textsc{Bool} \qquad \frac{x : \forall\vec{\alpha}.(\mathcal{C}_x \Rightarrow \alpha_x) \in \Gamma \qquad \gamma, \vec{\beta} \text{ new}}{\{\gamma =^{\ell} \alpha_x, x^{\ell}(\vec{\beta})\}; \Gamma \vdash x^{\ell} : \gamma}\ \textsc{Var}$$

$$\frac{\mathcal{C}_1; \Gamma \vdash e_1 : \alpha \qquad \mathcal{C}_2; \Gamma \vdash e_2 : \beta \qquad \gamma \text{ new}}{(\{\alpha =^{\ell} \textbf{fun}(\beta, \gamma)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2); \Gamma \vdash (e_1\ e_2)^{\ell} : \gamma}\ \textsc{App} \qquad \frac{\mathcal{C}; \Gamma, x : \alpha_x \vdash e : \beta \qquad \gamma \text{ new}}{(\{\gamma =^{\ell} \textbf{fun}(\alpha_x, \beta)\} \cup \mathcal{C}); \Gamma \vdash (\lambda x.e)^{\ell} : \gamma}\ \textsc{Abs}$$

$$\frac{\mathcal{C}_1; \Gamma \vdash e_1 : \alpha \qquad \mathcal{C}_2; \Gamma \vdash e_2 : \beta \qquad \mathcal{C}_3; \Gamma \vdash e_3 : \delta \qquad \gamma \text{ new}}{(\{\alpha =^{\ell_1} \textbf{bool}, \beta =^{\ell_2} \gamma, \delta =^{\ell_3} \gamma\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3); \Gamma \vdash \texttt{if } e_1^{\ell_1} \texttt{ then } e_2^{\ell_2} \texttt{ else } e_3^{\ell_3} : \gamma}\ \textsc{Cond}$$

$$\frac{\mathcal{C}_1; \Gamma \vdash e_1 : \alpha_1 \qquad \mathcal{C}_2; \Gamma, x : \forall\vec{\alpha}.(\mathcal{C}_1 \Rightarrow \alpha_1) \vdash e_2 : \alpha_2 \qquad \vec{\alpha} = fv(\alpha_1) \setminus fv(\Gamma) \qquad \vec{\beta}, \gamma \text{ new}}{(\{\gamma =^{\ell} \alpha_2, x^{\ell}(\vec{\beta})\} \cup \mathcal{C}_2); \Gamma \vdash (\texttt{let } x = e_1 \texttt{ in } e_2)^{\ell} : \gamma}\ \textsc{Let}$$

Fig. 1: Typing rules for the OCaml fragment

Expressions	$e ::= x$	variable
	$\mid v$	value
	$\mid e\ e$	application
	$\mid$ **if** e **then** e **else** e	conditional
	$\mid$ **let** $x = e$ **in** e	let binding
Values	$v ::= n$	integer
	$\mid b$	boolean
	$\mid \lambda x.e$	abstraction
Monotypes	$\tau ::= g \mid \alpha \mid \textbf{fun}(\tau, \tau)$	
Polytypes	$\sigma ::= \tau \mid \forall\alpha.\sigma$	

Fig. 2: Idealized OCaml Fragment

Loc Index	$i ::= n$	
Weight	$\omega ::= n$	
Source Range	$\ell ::= line; col - line; col$	
Location	$L ::= i\ \ell$	no weight given
	$\mid i\ \ell\ \omega$	weight given
Constraint	$C ::= i\ \tau_1 = \tau_2$	equality
	$\mid i\ x(\vec{\beta})$	instantiation
Scheme	$S ::= i\ x(\vec{\alpha})\ \vec{C}$	
IR	$R ::= \vec{L}\ \vec{S}\ \vec{C}$	

Fig. 3: IR Grammar

support future work on newer versions of OCaml, we ported just the EasyOCaml constraint generation framework to be a stand-alone OCaml project depending on the `ocaml-base-compiler` package [21]. Since this package does not include other features of EasyOCaml, it is significantly easier to port it to new versions of OCaml.

D. Intermediate Representation (IR)

The IR consists of three sets: a set of program source ranges, a set of type schemes, and a set of constraints. Program locations may optionally be annotated by weights. Weights of zero correspond to hard constraints. Whitespace is completely ignored. The complete expression grammar is shown in Figure 3. In constraints, τ refers to a monotype from Figure 2.

The "Loc Indices" i must be distinct and essentially name the source ranges. Throughout the constraint (resp. schemes) portion of the IR, the indices are used to encode the source range where the constraint (resp. schemes) was created. Later, the encoder will use the locations to embed the shape of the AST into the encoding.

Each constraint scheme S corresponds to a variable x and its associated type scheme $\forall\bar{\alpha}.(C_x \Rightarrow \tau_x)$. In particular, the scheme relates the name x, the quantified variables $\bar{\alpha}$, and the constraint set C_x. There is no special mention of α_x. The relationship between the scheme and α_x is encoded in how α_x (and its instantiations) appear in the constraints. Regardless, for human readability, Tyro always places α_x at the end of $\bar{\alpha}$.

Every constraint is either an equality of OCaml monotypes (which can be type variables), or an instantiation constraint. Instantiation constraints **can** appear inside schemes, which occurs whenever a polymorphic function is used within a polymorphic definition.

Tyro generates the constraint portion of the IR from the constraint set $\mathcal{C}$ of the top-level invocation of the constraint generation routine. Schemes are accumulated on the side, and always emitted. Location annotations are treated similarly.

The use of an intermediate representation is not necessary to the functionality of the system. However, it offers several advantages. Primarily, unlike the SMT encoding, the IR is human-writable and indeed human-readable given a bit of time. The final encoding, in contrast, is deeply nested and littered with information about the AST structure, making it quite difficult to read or write. Inspecting these intermediate files was invaluable for debugging constraint generation, and writing them by hand was further valuable for debugging the SMT encoder. This separation makes it easier to trust the correctness of the constraint generation and encoding steps.

Additionally, the use of an IR promoted modularity and reusability between the components. While working on Tyro, we were able to mix-and-match different methods of encoding the IR, without making any changes at all to the constraint generator. Similarly, we were able to redesign a significant portion of the constraint generator without any fear of breaking the encoder.

E. SMT Encoder

The SMT encoding step translates the intermediate representation to SMT-LIB [18] code. The only extension required to SMT-LIB 2.6 is vZ, for MaxSMT [22]. A Tyro run on the example from Section 2.2 of [16] can be seen in Figure 4. In particular, our SMT encoding is in Figure 4d.

Type schemes become SMT interpreted functions for the solver to instantiate on-demand. Equality constraints on types are encoded directly as equality constraints in the theory of inductive datatypes, using a `Type` sort to represent OCaml types. The `Type` sort is as described for MinErrLoc [16].

Type variables are encoded with a "-" in front of their name, to avoid conflicts with scheme names. This serves the same purpose as the single quote ("tick") in OCaml source code, but ticks are not allowed at the start of an SMT variable name.

The SMT encoding of constraints incorporates information about the AST structure. The enumeration of source locations is examined to recover an "AST forest." Each interval in the enumeration becomes a (possibly indirect) child of every interval that contains it. The result is a forest of program locations. In practice, this forest contains one tree for every top-level expression or let binding or in the program.

Consider the program fragment:

```
let x = "hi" in not x          (Ex.)
```

There are 5 source ranges in this fragment, shown in Figure 4b. If the MaxSMT solver decides to remove the entire fragment (location ℓ_0, the root of the tree), then all four of the other subfragments are necessarily removed as well. The weight of this decision must be determined only by the weight of location ℓ_0, even though all of its children are also being removed.

Therefore, for the fragment above, we encode a constraint C at location ℓ_3 as

$$\ell_0 \Rightarrow (\ell_4 \Rightarrow (\ell_3 \Rightarrow C))$$

The location variables ℓ_i are (softly) asserted directly with their weight. For example, with this fragment, we have

```
(assert-soft ℓ0 :weight 5)
(assert-soft ℓ3 :weight 1)
(assert-soft ℓ4 :weight 3)
```

The decision to remove location ℓ_0 (by setting the SMT variable ℓ_0 to `false`) now carries a cost of 5. The constraint C would no longer be active, even if ℓ_3 and ℓ_4 were still set to `true`.

All constraints are encoded in this way, starting at the root of an AST. Paths are combined, such that all of the constraints associated with a particular top-level statement are encoded into a single `assert` form. For example, two constraints C_1, C_2 at location ℓ_4 would be represented by only one copy of the above constraint encoding, with $C = C_1 \land C_2$. We apply this in a nested fashion, so each assertion consists of many nested implications and constraints. The constraints contained in a type scheme are also encoded this way, but are placed into an SMT "defined function" rather than using an `assert` form. The assertion tree for a scheme is rooted at the AST node which defined the scheme. In the case of a distant reference to a let-bound variable, this ensures that the instantiation constraint's implied constraints are disabled if (any parent of) the let binding is removed.[4]

The encoder provides MinErrLoc's weighting heuristic as a default if weights are not provided. Each node in the AST forest is assigned a weight equal to the size of the sub-AST rooted at that node. In example (Ex.) above, location 4 is assigned weight 3, ensuring that removing location 4 is more costly than removing both location 2 and 3 (which have a cumulative weight of 2). In its current configuration, Tyro uses the default weight for almost all locations.

Inspecting Figure 4c, the IR illustrates the change from MinErrLoc's encoding to Tyro's: the constraint `'x = string` came from a let binding, and is now part of a scheme. When the script in Figure 4d is run through Z3 [15], location ℓ_1 is identified as the error source.[5]

Taking advantage of the modularity offered by our design, we also implemented another SMT encoding which avoids deeply-nested implications. The shallower encoding appears to help the SMT solver in some cases. When the minimal cost is high, the shallower encoding can result in error sources that are not actually minimal. Empirically, however, almost all error sources for programs in our dataset had low costs. The MinErrLoc artifact employs the same alternate encoding, so we used it while evaluating Tyro.

F. Backend

The output of the encoder is an SMT-LIB script. The scripts are compatible with any SMT solver that supports at least SMT-LIB 2.6 [18] and the vZ extension for MaxSMT [22]. Tyro uses Z3 by default. The output of the SMT solver is processed to extract the minimum error source.

IV. Evaluation

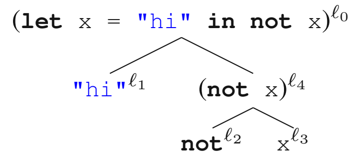
The MinErrLoc approach was evaluated for performance on a dataset of 356 programs collected from a programming course [16]. We collected several thousand programs from a programming course [23] and took a random sample of 500

[4]Instantiation constraints for a scheme can arise in only two cases: the binding is local, and the reference is a child of the binding in the AST; or the binding is top-level, and therefore the scheme's root is also the AST root.

[5]There are 3 minimal error sources for this program: $\{\ell_1\}$, $\{\ell_2\}$, and $\{\ell_3\}$.

$\texttt{let}^{\ell_0}\ \texttt{x} = \texttt{"hi"}^{\ell_1}\ \texttt{in}\ (\texttt{not}^{\ell_2}\ \texttt{x}^{\ell_3})^{\ell_4}$

(a)



(b)

```
<locations omitted>
---
0 x('x) {
  1 'x = string
}
---
0 x('x0)
2 'l2 = bool -> bool
3 x('x1)
4 'l2 = 'x1 -> 'l4
```

(c)

```
(declare-datatype Type
  ((string) (bool) (-> (->.1 Type) (->.2 Type))))
(declare-const ℓ0 Bool)(assert-soft ℓ0 :weight 5)
(declare-const ℓ1 Bool)(assert-soft ℓ1 :weight 1)
...
(declare-const -x0 Type)(declare-const -l2 Type) ...
(define-fun x ((-x Type)) Bool
  (=> ℓ0 (=> ℓ1 (= -x string))))
(assert
  (=> ℓ0 (and (x -x0)
    (=> ℓ4 (and (= -l2 (-> -x1 -l4))
      (=> ℓ2 (= -l2 (-> bool bool)))
      (=> ℓ3 (x -x1)))))))
(check-sat)(get-objectives)(get-value (ℓ0 ℓ1 ℓ2 ℓ3 ℓ4))
```

(d)

Fig. 4: A sample run of Tyro.
(a) an ill-typed program from [16] with locations annotated; (b) labeled program AST;
(c) simplified intermediate representation; (d) SMT encoding.

programs each from three different assignments for a total of 1500 programs. Programs were only selected if they could be parsed, but did not compile. Of the 1500 programs selected, approximately 70 contained localized errors other than type mismatches and were discarded. As Tyro is an experiment in delayed instantiation, we focused our evaluation on delayed instantiation. Though constraint slicing and preemptive cutting are shown to be both effective and simple to implement by MinErrLoc, our evaluation of Tyro did not use them.

A. *Timing*

Our statistics for timing Tyro are shown in Figure 5. Experiments were conducted on an Intel(R) Core(TM) i7-8550U CPU with four 1.80 GHz cores. Our experiments only used a single core for each instance of Tyro, but ran Tyro on several programs simultaneously. Tyro was run with a 100 second timeout, which excluded a further 40 programs, all from the same homework assignment. The statistics shown are for the remaining 1388 programs, in a format easily compared with MinErrLoc's evaluation in Figure 11 of Pavlinovic et al. [16].

We split our dataset into groups based on program length in lines of code. The number in parentheses is the number of programs in that group. The number of equality constraints, the minimum error source weight, and the time to run Tyro were recorded for each program. Note that the number of equality constraints cannot indicate how many times instantiation constraints will cause those equality constraints to be copied; therefore it is only a lower bound on the complexity of the MaxSMT problem.

In all groups, the constraint counts generated for our programs are significantly higher than those for MinErrLoc's evaluation. This suggests a difference in the typical structure of the programs which makes the evaluations hard to compare. Despite the slower processor used in our experiments and the generally higher constraint counts, we exhibit remarkably similar minimum and median execution times. Approximately 2.9% of programs evaluated timed out, and our maximum execution times are similar, though again slower, to those of MinErrLoc's evaluation for groups with similar constraint counts.

Our results are therefore promising. Our evaluation largely affirms that of MinErrLoc, on a significantly larger dataset.

One potential explanation for the lack of significant improvement is to consider how the SMT solver proceeds with instantiation constraints. As noted by MinErrLoc, the time spent copying constraint sets for instantiation during constraint generation is significant [16]. By delaying this work to the SMT solver, we create opportunities for the solver to recognize that an instantiation is not necessary at all. But we also risk that the SMT solver may perform a single instantiation many times. Given the cost of instantiations, the risks may outweigh the benefits for the version of Z3 used. This may improve in the future as solvers improve. We posit that SMT scripts generated by Tyro may make good benchmarks for MaxSMT solvers.

B. *Localization Accuracy*

We first took a random sample of 50 programs from our data set and labeled the true error source by hand. 8 of the programs were discarded because we could not decide which

Group		Constraints			Weight			Time (s)		
		min	**med**	**max**	**min**	**med**	**max**	**min**	**med**	**max**
0-50	(5)	44	63	72	1	1	3	0.02	0.11	0.16
50-100	(57)	96	276	990	1	2	35	0.08	0.68	2.93
100-150	(659)	111	532	1741	1	2	33	0.09	2.62	85.18
150-200	(449)	399	976	2341	1	3	23	0.84	17.80	87.86
200-250	(55)	696	1463	2702	1	2	18	1.53	10.50	89.43
250-300	(13)	633	1514	3039	1	1	6	2.94	7.58	86.31
300-350	(5)	1073	1516	2690	1	2	3	8.52	14.10	50.44

Fig. 5: Statistics for Tyro execution on whole programs

Tyro	OCaml	# of outcomes
hit	hit	5
hit	close	6
hit	miss	3
close	hit	3
close	close	20
close	miss	1
miss	hit	3
miss	close	0
miss	miss	1

Fig. 6: Accuracy on expert-labeled programs

Tyro	OCaml	# of outcomes
hit	hit	430
hit	close	9
hit	miss	15
close	hit	39
close	close	11
close	miss	2
miss	hit	113
miss	close	3
miss	miss	25

Fig. 7: Accuracy on automatically labeled programs

of several error sources were most likely. The comparison of Tyro's accuracy versus `ocamlc`'s on these programs is shown in Figure 6. They are formatted for easy comparison to Figure 8 of the MinErrLoc analysis [16]. Regions were marked as "hit" if they exactly matched the true error source. If the region was close enough for a (novice) programmer to easily understand the true problem, the region was marked as "close." Otherwise, it is marked "miss."

Our expert-labeled evaluation uses a larger dataset than MinErrLoc's expert-labeled evaluation (40 programs versus 20) and displays almost identical proportions of outcomes. This reaffirms the small-scale evaluation results of MinErrLoc.

We reviewed the one program where both Tyro and OCaml missed. It is an especially tricky case where the true error source contains two program locations, and their relationship is partially obscured by the programmer's mistake. Tyro and OCaml report adjacent program locations (both of weight 1), neither of which are members of the true error source.[6] In the other 41 programs, either Tyro or OCaml identify the true error source.

We experimented with automatic methods for evaluating localization accuracy, using a similar approach to [24]. We compare the region(s) reported by localization to the region(s) that students actually modified to fix a type error. For each of the 1388 programs in our random sample, we determined if the successive code sample from the same student compiled successfully. We recover the regions that the student modified using Difftastic [25], a structural differencing tool, and then removed programs where Difftastic reported a high portion of the file had been rewritten. In this manner, we collected 647 data points. We then classified the identified regions in an automated manner similar to the expert-labeled evaluation. Exact matches were marked as "hit", other forms of (possibly partial) overlap or shared endpoints were marked as "close", and anything else was marked as a "miss." Notably, consider an application such as `f x`. If the student modified `x`, but the identified region was `f`, these intervals are considered to share an endpoint and are marked "close." This situation appears to be quite common, as does the reverse.

Unfortunately, this approach suffers from a major source of bias: because the students fixing the program only had access to error messages from OCaml, they were far more likely to modify the region of code indicated by OCaml (which is always a member of some error source). This bias is clearly seen in the results in Figure 7.

As part of typical homework assignments in our course, students write their own test cases. These test cases are formatted as lists of input-output pairs. One test case was part of the given code. For some problems, the given test case was correct. For other problems, students were supposed to fix an incorrect test case. We inspected a random sample of the 113 programs where Tyro missed but OCaml hit. In approximately 70% of the sampled programs, the type error was due to malformed test cases. The students wrote several test cases containing `int`s where `float`s were expected, or vice versa. Because the students wrote several cases after the one given case, the minimum error source is always the given test case.

[6]However, if both OCaml and Tyro's reported locations are made hard constraints, the true error source becomes a minimum error source.

But the given test case comes first, so OCaml reports the mismatch on the cases written by the student. Tyro "misses" for these programs because the students followed OCaml's advice – even when that advice was incorrect.

This demonstrates the subjectivity of the type error localization problem, and provides evidence that type annotations should be used judiciously to guide students. If a top-level type annotation had been included for the test cases and set as a hard location, Tyro and OCaml would both identify the incorrect test cases.[7]

Considering this bias, Tyro appears remarkably accurate despite the fact that we are using the "relatively simplistic" weighting heuristic of AST size. This again reaffirms the potential of the MaxSMT localization approach.

Out of the 647 programs evaluated, either Tyro or OCaml identify the true error source in over 96% of cases. This is similar to our observation from the expert-labeled evaluation. Therefore, we conclude that reporting localizations from Tyro alongside OCaml's error report would be an effective, accurate diagnostic for programmers.

V. Related Work

MinErrLoc [16] first demonstrated that type error localization problems can be efficiently expressed as Partial Weighted MaxSMT problems. They recognize the issues associated with polymorphic types, but do not simplify them. They propose two algorithms to improve the situation: Lazy Quantifier-Based Instantiation, and Lazy Unification-Based Instantiation. Tyro implements Lazy Quantifier-Based Instantiation.

Other tools have also begun using (Max)SMT solvers for type inference problems. Typpete [26] uses a MaxSMT solver to infer type annotations to be added to Python programs. Typpete additionally had to solve the challenge of encoding subtyping constraints. Similar ideas were discussed in the presentation of MinErrLoc. We believe our architecture could be leveraged to tie these ideas together and create localization tools for languages like Java or Haskell.

Mycroft [14] takes a different approach to localization by heuristic minimization. Rather than reducing localization to MaxSMT, Mycroft is a solver dedicated to minimizing error sources in type inference problems. It is generalized over the type system being used and requires an inference engine for that system. The Mycroft algorithm is very similar to MaxSMT algorithms based on "Unsatisfiable Cores" [27]. Mycroft's ability to use a dedicated typechecking engine means it can avoid issues like the polymorphic constraint blowup seen in MinErrLoc and Tyro. Unfortunately, Mycroft does not benefit from frequent improvements to the MaxSMT state-of-the art.

Zhang and Myers have previously reduced localization problems to finding certain types of paths in a graph [28]. They apply Bayesian methods to guess which source location to blame for the faulty paths. This work was further developed to support advanced type system features like *type classes* in Haskell [29] and an implementation, SHErrLoc, is available [30]. Their graphs did not encode the "flow" of typing information during the inference process. A recent approach, HM^{ℓ}, takes inspiration from subtyping systems to express the way that typing information flows through the inference process [7]. Rather than heuristically producing a localization guess, HM^{ℓ} error messages contain a detailed flow diagram containing all of the source locations participating in the error. They report that this can lead to "information overload," however, it is a promising new view on the problem.

[7]Such annotations are recommended by Pavlinovic et al. [16], but unfortunately we did not have control over the content of the assignments.

VI. Future Work

We have observed several potential avenues for future work on Tyro or other tools. The most obvious is perhaps to improve the weighting heuristic.

While Tyro implements the Lazy Quantifier-Based Instantiation proposal from [16], a unification-based algorithm was also proposed. The proposed algorithm makes several calls to the SMT solver, and requires changing the constraints related to polymorphic variables on every call to the solver. This would be a considerable challenge for the architecture of MinErrLoc. However, because Tyro separates constraints related to polymorphic variables from other constraints, it seems the algorithm could be implemented on top of Tyro in a relatively straightforward fashion, which we intend to explore in future work.

For future work on MaxSMT solvers, we believe that MaxSMT scripts generated by Tyro have potential as benchmarks.

VII. Conclusion

Tyro is a modernization of the MinErrLoc MaxSMT approach to type error localization. Our evaluation reaffirms the accuracy and performance potential of the approach using a larger dataset. Our evaluation for accuracy indicates that a less simplistic metric than AST size may perform better, at least on student programs. Regardless, our evaluation shows that the combination of Tyro and OCaml already exhibits an accuracy above 96%.

Tyro's modular design makes it easy to experiment with modifications to various aspects of the system. Indeed, we experimented with some variations on the AST size heuristic,[8] completely rewriting the constraint generation frontend, and several SMT encodings. While incorporating lazy quantifier-based instantiation did not immediately improve the performance of the approach, we believe Tyro's architecture will allow it to serve as a testbed for future work on MaxSMT-based localization.

VIII. Acknowledgements

This work was supported by the Social Sciences and Humanities Research Council (SSHRC), the OCaml Software Foundation, and the Canada CIFAR AI Chair Program.

[8]MinErrLoc also incorporates at least one such variation.

References

[1] OCaml Foundation, "OCaml." [Online]. Available: https://ocaml.org/

[2] B. Wu and S. Chen, "How type errors were fixed and what students did?" in *Proceedings of the ACM on Programming Languages*, vol. 1, OOPSLA, Oct. 2017, pp. 105:1–27.

[3] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2007, p. 425–434.

[4] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning Haskell," in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Aug. 2003, pp. 62–71.

[5] M. Wand, "A semantic prototyping system," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, Jun. 1984, p. 213–221.

[6] M. Wand, "Finding the source of type errors," in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1986, pp. 38–43.

[7] I. Bhanuka, L. Parreaux, D. Binder, and J. I. Brachthäuser, "Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference," in *Proceedings of the ACM on Programming Languages*, vol. 7, OOPSLA2, Oct. 2023, pp. 431–459.

[8] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala, "Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis," in *Proceedings of the ACM on Programming Languages*, vol. 1, OOPSLA, Oct. 2017.

[9] C. Geng, H. Ye, Y. Li, T. Han, B. Pientka, and X. Si, "Novice Type Error Diagnosis with Natural Language Models," in *Programming Languages and Systems*, Dec. 2022, pp. 196–214.

[10] M. Sulzmann, M. Muller, and C. Zenger, "Hindley/Milner style type systems in constraint form," Tech. Rep., Oct. 1999.

[11] F. Pottier and D. Rémy, "The Essence of ML Type Inference," in *Advanced Topics in Types and Programming Languages*, Jan. 2005, pp. 389–489.

[12] O. Kiselyov, "Efficient and Insightful Generalization." [Online]. Available: https://okmij.org/ftp/ML/generalization.html

[13] M. Odersky, M. Sulzmann, and M. Wehr, "Type inference with constrained types," *Theory and Practice of Object Systems*, vol. 5, no. 1, pp. 35–55, Jan. 1999.

[14] C. Loncaric, S. Chandra, C. Schlesinger, and M. Sridharan, "A Practical Framework for Type Inference Error Explanation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 51, no. 10, Oct. 2016, p. 781–799.

[15] L. De Moura and N. Bjørner, "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*, Mar. 2008, p. 337–340.

[16] Z. Pavlinovic, T. King, and T. Wies, "Finding minimum type error sources," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, Oct. 2014, pp. 525–542.

[17] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of Computer Aided Verification - 23rd International Conference*, vol. 6806, Jul. 2011, pp. 171–177.

[18] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017. [Online]. Available: www.SMT-LIB.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf

[19] C. Barrett, I. Shikanian, and C. Tinelli, "An Abstract Decision Procedure for a Theory of Inductive Data Types," *Journal on Satisfiability, Boolean Modeling, and Computation (JSAT)*, vol. 3, pp. 21–46, Jul. 2007.

[20] B. Becker, C. Haack, and J. B. Wells, "EasyOCaml." [Online]. Available: http://easyocaml.forge.ocamlcore.org/

[21] X. Leroy, "ocaml-base-compiler." [Online]. Available: https://ocaml.org/p/ocaml-base-compiler/

[22] N. Bjørner and P. Dung, "vZ - Maximal Satisfaction with Z3," in *Proceedings of the 6th International Symposium on Symbolic Computation in Software Science*, Dec. 2014.

[23] A. Ceci, H. C. A. Tavante, B. Pientka, and X. Si, "Data Collection for the Learn-OCaml Programming Platform: Modelling How Students Develop Typed Functional Programs," in *SIGCSE '21: The 52nd ACM Technical Symposium on Computer Science Education*, Mar. 2021, p. 1341.

[24] E. L. Seidel, "Data-driven techniques for type error diagnosis," Ph.D. dissertation, University of California, San Diego, USA, 2017. [Online]. Available: http://www.escholarship.org/uc/item/59s4h4pv

[25] W. Hughes, "Difftastic," 2021. [Online]. Available: https://github.com/wilfred/difftastic

[26] M. Hassan, C. Urban, M. Eilers, and P. Müller, "MaxSMT-Based Type Inference for Python 3," *Computer Aided Verification: 30th International Conference*, pp. 12–19, Jul. 2018.

[27] J. Marques-Silva and J. Planes, "Algorithms for Maximum Satisfiability using Unsatisfiable Cores," in *2008 Design, Automation and Test in Europe*, Mar. 2008, pp. 408–413.

[28] D. Zhang and A. C. Myers, "Toward General Diagnosis of Static Errors," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2014, pp. 569–581.

[29] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton Jones, "Diagnosing type errors with class," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 50, Jun. 2015, pp. 12–21.

[30] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones, "SHErrLoc: A Static Holistic Error Locator," *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 4, Aug. 2017.

[31] R. Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dec. 1978.

Appendix

A. Polymorphic Types

OCaml's type system assigns types to all expressions, for example an integer literal like `5` has type `int`. Function types are written with an arrow, for example a fibonacci function might have type `int` $\to$ `int`.

Consider an identity function, defined with

```
let id x = x;;
```

What ought to be the type of this function? If we infer a type like `int` $\to$ `int` (which is certainly sound), we won't be able to use the function with booleans, or vice versa. If we assign it the type $\alpha \to \alpha$, where α is a (monomorphic) type variable, we still have a problem: we can use the function at `int` *or* at `bool`, but not both. In fact, this function is frequently passed as an argument to higher-order functions, and therefore it is common to have it used at many different types throughout a program.

The solution taken by "Hindley-Milner type systems" [31] allows *polymorphic* types. We might express the true type of `id` as $\forall \alpha . \alpha \to \alpha$. Quantifying over the type variables in a type is called *generalization*. Whenever the variable `id` is referred by the program, a new monomorphic type variable will be created to represent α for that specific instance, a process called *instantiation*. Only values bound with a **`let`** binding are generalized – notably, lambda abstractions are *not* generalized (unless they are later bound by a **`let`**).

Polymorphic types are a major challenge for type error localization [6], [16], in large part because the generalization and instantiation processes make it difficult to tie a type mismatch from *outside* of the definition of a **`let`** binding back to a source in the body of the binding.

B. Classical Type Inference

The goal of type inference is to assign a type to every (sub)expression in the program, thereby ensuring that the program is type-safe, but without requiring any annotations from the programmer.

The classical type inference algorithm described in [31] proceeds via structural recursion on the program AST. Each node of the AST corresponds to a (sub)expression of the program. We use the kind of each subexpression to infer the "shape" of its type – lambda abstractions must have a function type, boolean literals must have the `bool` type, etc. Any unknown information in the inferred shape, such as the input and output types of a function type, are filled with (monomorphic) type variables. When these type variables correspond to the type of a named program variable, this relationship is stored in a *context*.

As we recurse through the AST, we may discover relationships between some of the inferred shapes. For example, when a lambda abstraction is applied to an expression e, we learn that the abstraction's input type must match the type of e. We use this information to refine the type variables in both types through a process called *unification*. Unification "solves for" some or all of the type variables in both types.

A second approach to refining types is to store all of the discovered relationships as *typing constraints* [10]. These constraints can be generated for the whole program, and then later fed into a constraint solver all at once. We must use such a constraint-based algorithm; see Section II-A for why.

Formal Methods in Computer-Aided Design 2024

Context Pruning for More Robust SMT-based Program Verification

Yi Zhou, Jay Bosamiya, Jessica Li, Marijn J. H. Heule, Bryan Parno
Carnegie Mellon University, Pittsburgh, PA, USA
{yeet,jaybosamiya,jgli,marijn,parno}@cmu.edu

***Abstract*—SMT solvers provide powerful proof automation for program verification. However, relying on SMT solvers also leads to proof instability, where a previously successful proof may fail after the developer makes trivial modifications to the source program. Such instability is a major headache for developers, but the causes and potential mitigations for it have received limited attention. In this study, we find that irrelevant query context accounts for 78% of the instability in existing program-verification query sets. As a result, we design SHAKE, a novel technique that leverages the structure in program-verification SMT queries in order to filter out irrelevant context from such queries. SHAKE is the first SMT-level technique that targets instability, and we implement it as a pre-processing step for SMT solvers. We evaluate SHAKE on real-world, large-scale query sets, and we find that it leads to large reduction in context and a 29% and 41% improvement in query stability on Z3 and cvc5, with minor performance overhead.**

I. Introduction

Satisfiability Modulo Theories (SMT) solvers play a crucial role in *automated program verification*, since verification-oriented languages (e.g., Dafny [1] or F* [2]) often translate program source code and specifications into verification conditions [3], [4] that they encode as SMT queries [5]. Essentially, each SMT query states that the code adheres to its specifications, and the SMT solver (e.g., Z3 [6] or cvc5 [7]) checks if this statement holds. The solvers obviate many manual proof steps, simplifying the verification of large code bases [8]–[14].

Unfortunately, SMT-based program verification is not necessarily robust. Notably, the approach is susceptible to *proof instability* [15], where trivial changes to the program cause spurious verification failures. For instance, the SMT solver may reject a previously-verified program after the developer renames a variable, even though the program's semantics clearly did not change. Faced with such a proof failure, the developer may need to tediously provide manual proof steps to guide the solver back on track [16], which arguably defeats the purpose of automation. To the frustration of practitioners, instability has been a long-standing problem [15], [17]–[23]. While instability is pervasive in practice [15], its causes remain understudied, let alone its mitigation. Existing literature has pointed at several potential culprits [18]–[20], but these claims are anecdotal and lack quantitative evidence.

In this work, we explore the problem quantitatively and find that irrelevant query context is a *major contributor* to instability. Our experiments on unsatisfiable cores from a large-scale program-verification query set discover that typically 96%–99% of the assertions in a query do not remain in the unsatisfiable core—they are irrelevant to verification. More importantly, irrelevant assertions account for 78% of the observed unstable instances (§III).

Motivated by the findings, we propose a novel SMT context-pruning technique, named SHAKE, to improve stability. We base SHAKE on the insight that program-verification tasks are typically automated theorem proving (ATP) [24] tasks, meaning that the verification queries are each composed of a goal assertion along with axiom assertions. SHAKE triages the axioms with respect to the goal and prunes the less relevant axioms.

While SMT solvers are built for constraint-solving, adopting a theorem-proving perspective helps improve stability. We implement SHAKE as a preprocessor, and evaluate it on large-scale program-verification query sets from the Mariposa study [15]. We find that SHAKE typically reduces the context by 3–10$\times$. Moreover, we show that SHAKE can mitigate instability on Z3 by 29% and on cvc5 by 41%. SHAKE imposes little runtime overhead, even improving the number of solved instances on cvc5 by 73% in one benchmark and 8% overall.

In summary, we make the following contributions.

- We empirically show that irrelevant context is a major source of instability in program-verification queries.
- We propose a novel pruning technique, SHAKE, based on a theorem-proving view of program verification.
- We show that SHAKE reduces instability by 29%–41% on existing query sets, with only minor overhead.

To facilitate research on context pruning and instability mitigation, our source code and query sets are all available at https://github.com/secure-foundations/mariposa.

II. Background

Formal verification provides strong guarantees about program properties such as security and functional correctness. In recent years, academia has made notable progress in verifying large-scale systems [8]–[14], [25]–[28]. Industry has also adopted verification in certain mission-critical scenarios [29]–[31]. In particular, automated verification languages have gained popularity, exemplified by Dafny and F*, which are maintained by Amazon Web Services and Microsoft Research respectively.

These automated verification languages are powered by SMT solvers. Typically, a language's verification condition generator (VCG) encodes the source program into a logical formula, which states that the program's specification holds; i.e., the program is correct. If the SMT solver reports the negation of

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_12

the formula to be *unsatisfiable*, the program's specification is never violated, and thus the program verifies. However, since program properties are generally undecidable, the solver cannot guarantee that it will verify every correct program.

This incompleteness then leads to the phenomenon of *proof instability*, where a previously successful verification spuriously fails after trivial modifications to the source program. This happens because source-level changes obligate the VCG to create a new query for the SMT solver. Due to incompleteness, the solver may succeed on an old version of the query but may fail on the new one, even if the queries are semantically equivalent.

Instability is a major headache for developers. For individuals, it disrupts their incremental development process by diverting them from their main development tasks. For teams, instability is even more problematic, as instability may only appear when concurrent changes to the source code are merged.

In light of this problem, the Mariposa project [15] aims to quantify instability in SMT-based program verification. For a given SMT query-solver pair (q, s), the Mariposa tool outputs a stability category: `stable`, `unstable`, or `unsolvable`. In some cases the status may be `inconclusive`, which indicates that Mariposa does not have sufficient statistical power to confidently assign a category.

The Mariposa tool derives the stability status from the performance of s on q's mutants, which are semantically equivalent to q. Specifically, Mariposa creates the mutants by shuffling the assertions or renaming the symbols in q, as well as by reseeding the random number generator in s.

The Mariposa project experimented with large-scale program verification query sets. For this study, we use the Mariposa methodology to measure instability, and we also conduct our experiments on the Mariposa query sets. We exclude one query set, Komodo$_S$, from our study, which we discuss in §VII.

III. Query Context

In this section, we study the connection between query context and stability. We abstract an SMT query's context as a set of assertions, each introducing a constraint to the query. We then analyze each query's unsatisfiable core. Upon reaching an unsat result, the solver produces a core, which is a subset of the original assertions that the solver used to derive the unsat result. Thus, the solver-produced core serves as an oracle of relevant assertions, and what is excluded from the core can be considered irrelevant.

In §III-A, we describe our method to obtain unsat cores. In §III-B, we show that often only a tiny fraction of the assertions are relevant to verification success. In §III-C, we show that irrelevant context can be a major source of proof instability. In §III-D, we present a simple theorem-proving view of the query context and discuss how that view can help cut down on irrelevant assertions to improve stability.

A. Export the Unsatisfiable Core

In theory, we can export an unsat core by enabling an SMT solver's `produce-unsat-cores` option. In reality, obtaining an unsat core can sometimes be non-trivial, especially on unstable queries. Though uncommon, two types of problems may occur, so we document our workarounds here.

Unsuccessful Export. The solver might not be able to produce a core. There can be several reasons. First, the solver behaves differently depending on whether the core is requested or not. We have observed cases in which the solver returns `unsat` on a query, but returns `unknown` when core production is enabled. Second, the query itself might be unstable, meaning that the original query may fail, but some mutants of it may succeed. Third, a query might be completely unsolvable (regardless of mutations) with a particular solver version, but solvable with another.

In these cases, we perform Mariposa-style mutations to the query, attempting to obtain a core from any of the mutants. We then map the core from a successful mutant back to a core of the original query. If necessary, we also try the core export using different versions of the solver.

Incomplete Core. The solver might also produce a core query that is incomplete. Specifically, the solver might return `unsat` on the original query and successfully produce a core query; however, when given the core query, the solver fails to produce `unsat`, even with mutations applied to the core. This could be due to certain assertions that are necessary to the proof but missing in the core. Note that incompleteness here is not a strictly formal notion, since we do not have a ground truth for necessity.

When this happens, we apply a best-effort search to repair the core by adding assertions back to the core query, performing a bisection search to find a small addition of assertions that make the solver return `unsat` on the core. In practice, we find the incompleteness problem to occur more often with F* queries ($\sim 8\%$), and the core is typically only "missing" a small number (≤ 5) of assertions.

In summary, if the two issues above occur, we make a best-effort attempt to find a core query such that: its assertions form a subset of the original's, and it is sufficient for the solver to show `unsat`. We are successful in these attempts for all but a small fraction of the original queries. In that remaining fraction, we use the original query as the core query.

B. Most of the Context is Irrelevant

After acquiring an unsat core, we compare its context to the original. As shown in Figure 1, the original query context typically contains thousands of assertions. Using the assertion count as a proxy for the "size" of the context, we examine the *relevance ratio*:

$$\frac{\text{\# core assertions}}{\text{\# original assertions}} \times 100\%$$

Since an unsat core is a subset of the original query, the lower this ratio is, the less context is retained, and the more irrelevant context the original query has.

Figure 2 shows the CDFs of the relevance ratios for different projects. For example, on the left side lies the line for DICE$^\star_F$. The median relevance ratio (MRR) is 0.06%, meaning that

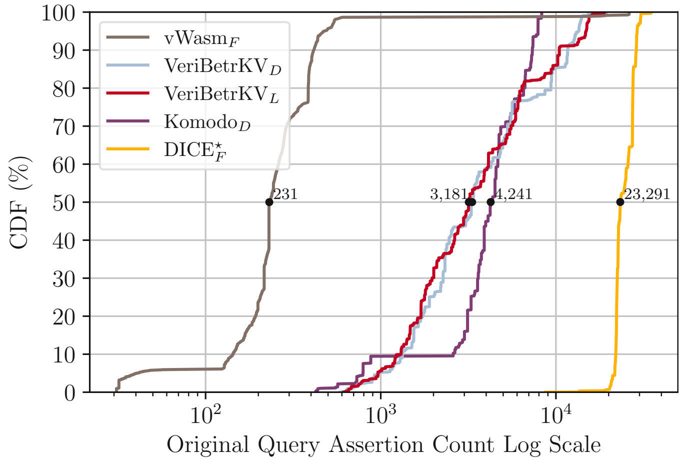


Fig. 1. **Original Query Assertion Count.** More to the right means larger query contexts, which may each contain thousands of assertions.

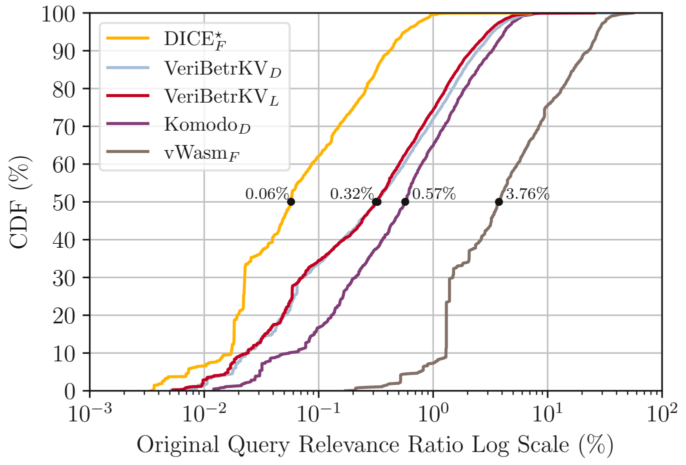


Fig. 2. **Original Query Context Relevance.** More to the left means more irrelevant contexts. Typically, the vast majority of an original query context is irrelevant.

for a typical query in the project, only 0.06% of the context is relevant. In vWasm$_F$, the MRR is 3.76%, which is almost an of order of magnitude higher than the other projects. We attribute this to the manual context tuning by the authors of vWasm$_F$, who explicitly documented the tedious effort [14], [15]. Nevertheless, if we consider the complement of the relevance ratio, typically 96.23–99.94% of the context is irrelevant, even considering vWasm$_F$.

C. Irrelevant Context Harms Stability

Given the significant amount of irrelevant context, we further analyze how that impacts stability. Here we compare and contrast the stability of the original queries and their cores. Recall the Mariposa stability status for a query-solver pair can be one of `unsolvable`, `unstable`, `stable`, or `inconclusive`. Given an original query q and its core q_c, we introduce the following two metrics:

- **Preservation**: given that q is `stable`, the probability that q_c remains `stable`.
- **Mitigation**: given that q is `unstable`, the probability that q_c becomes `stable`.

We use the Mariposa tool [32] with Z3 version 4.12.5 in this experiment. In Figure 3, we list the number of original queries and the scores for solver-produced core. As an example, in the original Komodo$_D$ queries, 1,914 are stable and 93 are unstable. In its core counterpart, 99.4% of the stable queries remain stable, while 90.3% of the unstable ones become stable. vWasm$_F$ is the only case where the core has no mitigation effect. However, its original queries are rarely unstable. As we noted previously, vWasm$_F$ also starts with more relevant original context. Therefore, the stability of vWasm$_F$ can be explained by the manual tuning done by the original developers.

Project	Original		Solver-Produced Core	
	Stable	Unstable	Preservation	Mitigation
Komodo$_D$	1,914	93	99.4%	90.3%
VeriBetrKV$_D$	4,983	172	99.5%	64.5%
VeriBetrKV$_L$	4,999	256	99.6%	83.6%
DICE$^\star_F$	1,483	20	99.6%	90.0%
vWasm$_F$	1,731	4	99.7%	0.0%
Overall	15,110	545	99.5%	78.3%

Fig. 3. **Stability of Core Queries.** Typically an unsat core preserves the stability of the original query, and it mitigates instability in 78.3% of the unstable queries.

Generally, the solver-produced unsat core is highly likely to preserve query stability. Moreover, across all projects, 78.3% of the unstable instances can be mitigated by using the core. In other words, irrelevant context is **a major contributor to instability**. This result suggests a promising mitigation strategy of pruning irrelevant assertions. In the next section, we discuss the composition of query context and how it can inform context pruning.

D. Context Pruning is Axiom Selection

In §II, we offered an overview of the verification condition generator (VCG) in automated verification languages. Here we give a more formal treatment on how a VCG constructs the query context, along with an intuitive view of the query as a theorem-proving task and context pruning as axiom selection.

The VCG typically creates an SMT query per procedure[1] under verification. Given a procedure P, the VCG encodes a verification goal ψ, which is a formula stating that P is correct. $\neg\psi$ is then placed into the query as an assertion. In practice, the goal ψ is rarely self-contained, since P usually refers to other procedures or relies on language-level axioms. The VCG also includes these dependencies in the query. As a result, the query context is a constraint set $\Gamma = \{\neg\psi\} \cup \Gamma_A$, where $\Gamma_A = \{\varphi_1, ..., \varphi_n\}$ is a set of axioms.

The standard semantics of an SMT query is the satisfiability of the constraint set Γ. We can interpret the query as checking the validity of $\Gamma \vdash$ `false`, which is equivalent to $\Gamma_A \vdash \psi$.

[1]We generically refer to a function-like construct with pre/post-conditions as a procedure. It can be a function, method, lemma, etc.

Intuitively, this is a theorem-proving task, where the axioms in Γ_A are given to prove the verification goal ψ.

Through this view, the context pruning problem becomes an axiom selection problem, in which we choose a subset of axioms $\Gamma_R \subseteq \Gamma_A$ s.t. $\Gamma_R \vdash \psi$. The SMT solver usually takes the constraint-solving view of the query, where the relevance of an assertion is determined by its contribution to the unsatisfiability. As it turns out, the solver can also benefit from this theorem-proving perspective, where we define the relevance of the axiom assertions with respect to the goal.

IV. SHAKE

In this section, we introduce SHAKE, a pruning technique for SMT queries produced during program verification. At a high level, SHAKE takes a query as input and computes the distance from each axiom to the goal, indicating the relevance. More formally, the input to SHAKE is a set of constraints $\Gamma = \{\varphi_0, ..., \varphi_n\}$. For convenience, let $\varphi_0 = \neg\psi$, where ψ is the verification goal, while $\varphi_1, ..., \varphi_n$ are the axioms. The output of SHAKE is thus a map of distances:

$$dists = \{(\varphi_0 : 0), ..., (\varphi_n : d_n)\}$$

where the goal is at 0. SHAKE then prunes the axioms based on their distances. We first introduce a naive version of SHAKE, then progressively improve upon the design.

A. *The Naive* SHAKE *Algorithm*

In this version of SHAKE, we abstract a formula ϕ via the set of query-defined symbols it contains, denoted as SYMBOLS(ϕ). More precisely, the symbols are the functions, constants, and datatypes introduced by the query, excluding sorts, local variables, and built-in SMT-LIB functions: intuitively, ubiquitous functions like $<$ or `not` do not convey much information.

Alg. 1 shows the naive SHAKE algorithm. We first initialize a context symbol set S_{ctx} from the goal. We then select all axioms φ_i such that SYMBOLS(φ_i) intersects with S_{ctx}, on the theory that intersection conveys relevance. After scanning through all axioms in this round, we augment S_{ctx} with the symbols from the selected axioms. The update is delayed until the end of the round, so S_{ctx} remains the same during this scan. Otherwise, the scan order would affect the content of S_{ctx}, introducing a form of instability.

Applying this process repeatedly scores the distance of an axiom φ_i based on the round in which SYMBOLS(φ_i) first intersects with S_{ctx}. The outer iteration continues until we reach a fixed point. When there are unreachable axioms at the end, they are assigned a distance of round count plus one.

In practice, we find that naive SHAKE typically terminates after very few iterations, giving little differentiation between axioms. The problem arises because naive SHAKE is too eager in its expansion. Since we use symbol sets to abstract away formulas, a single complex axiom with a large symbol set can easily saturate S_{ctx}, ending the process quickly. In light of this problem, we refine the formula abstraction to handle quantifiers, which SHAKE expands lazily.

Algorithm 1 Naive SHAKE

```
procedure NAIVESHAKE($\Gamma = \{\varphi_0, ..., \varphi_n\}$)
    # assuming $\varphi_0$ is the goal
    $S_{ctx} \leftarrow$ SYMBOLS($\varphi_0$)
    $dists, round \leftarrow \{(\varphi_0 : 0)\}, 1$
    repeat
        $acc \leftarrow \emptyset$
        for $\varphi_i \in \Gamma$ do
            if $S_{ctx} \cap$ SYMBOLS($\varphi_i$) $\neq \emptyset$ then
                # check if $\varphi_i$ has been assigned a distance
                if $\varphi_i \in$ UNREACHED($dists, \Gamma$) then
                    $dists \leftarrow dists \cup \{(\varphi_i : round)\}$
                $acc \leftarrow acc \cup$ SYMBOLS($\varphi_i$)
        # update the symbol set after considering all $\varphi_i$
        $S_{ctx} \leftarrow acc \cup S_{ctx}$
        $round \leftarrow round + 1$
    until ISFIXEDPOINT($dists$)
    $max\_dist \leftarrow round + 1$
    for $\varphi_i \in$ UNREACHED($dists, \Gamma$) do
        # assign maximum distance to unreachable axioms
        $dists \leftarrow dists \cup \{(\varphi_i : max\_dist)\}$
    return $dists$
```

```
(declare-fun foo (Int) Int)
(declare-fun bar (Int) Int)
(declare-fun qux (Int) Int)
(assert (forall ((x Int))
            (! (< (foo x) (bar (qux x)))
              :pattern ((foo x))
              :pattern ((bar x)))))
```

Fig. 4. **Example SMT Assertion with Pattern**. The patterns are hints to the solver on when to instantiate the quantifier. In this example, either the pattern `(foo x)` or the pattern `(bar x)` should be matched.

B. SHAKE *with Quantifiers*

In the queries we study, quantifiers often come with *patterns* [33], [34]. Patterns are syntactic hints to the solver as to when a quantifier should be instantiated; if the patterns are not matched, the quantified body remains hidden. In this version of SHAKE, we use the available patterns to refine the notion of relevance for formulas.

In this version, we construct a **formula state** for a given formula ϕ. We denote this via INITFSTATE(ϕ), which augments ϕ with two fields:

- $\phi.S_{visible}$: the set of symbols in ϕ not under any quantifier.
- $\phi.qstates$: a list of **quantifier states**, constructed only from the outermost quantifiers in ϕ. The construction via INITQSTATE is lazy, meaning that any nested quantifiers are hidden under the outermost quantifier states.

Given a quantified formula ω, INITQSTATE(ω) creates a quantifier state containing ω and two additional fields:

- $\omega.patterns$: a list of symbol sets from the patterns.
- $\omega.\phi_{hidden}$: the quantified body, which remains uninitialized until expanded, including any nested quantifiers it may contain.

For example, in Figure 4, the list of pattern symbol sets is $[\{$`bar`$\}, \{$`foo`$\}]$, and the hidden body is the formula `(< (foo x) (bar (qux x)))`.

SHAKE is lazy when determining the relevance of a quantifier state, reflected in the TRYEXPAND procedure. Given a symbol set S, if none of the $\omega.patterns$ is a subset of S, the quantifier is irrelevant, and ϕ_{hidden} remains unexpanded (i.e., SHAKE ignores the symbols it contains). The subset condition is necessary because for an actual instantiation, all the symbols in a specific pattern must be present in S. Upon a match, TRYEXPAND creates a new formula state from its hidden body ϕ_{hidden}. We note that the quantifier is only expanded by one level of nesting via INITFSTATE.

```
procedure TRYEXPAND($\omega, S_{ctx}$)
    $relevant \leftarrow$ false
    # subset check needed to check for pattern match
    for $S \in \omega.patterns$ do
        if $S \subseteq S_{ctx}$ then
            $relevant \leftarrow$ true
    if $relevant$ then
        # create a new formula state from the hidden body
        INITFSTATE($\omega.\phi_{hidden}$)
        return SOME($\omega.\phi_{hidden}$)
    return NONE
```

SHAKE checks the relevance of a formula state φ as follows. Given a symbol set S, φ is relevant if $\varphi.S_{visible}$ intersects with S, or if any of the $\varphi.qstates$ is considered relevant. When SHAKE expands a quantifier state, the resultant formula state is merged into φ. This process is reflected in the FORMULARELEVANT procedure below.

```
procedure FORMULARELEVANT($\varphi, S_{ctx}$)
    $qstates' \leftarrow [\,]$
    $relevant \leftarrow S_{ctx} \cap \varphi.S_{visible} \neq \emptyset$
    for $\omega \in \varphi.qstates$ do
        $r \leftarrow$ TRYEXPAND($\omega, S_{ctx}$)
        # expansion may create a new formula state $\phi_{hidden}$
        if SOME($\phi_{hidden}$) $= r$ then
            # a trigger matches; merge $\phi_{hidden}$ with $\varphi$
            $qstates' \leftarrow qstates' + \phi_{hidden}.qstates$
            $\varphi.S_{visible} \leftarrow \varphi.S_{visible} \cup \phi_{hidden}.S_{visible}$
            $relevant \leftarrow$ true
        else
            # no match; no new formula state created
            $qstates' \leftarrow qstates' + [\omega]$  # keep the quantifier state
    $\varphi.qstates \leftarrow qstates'$
    return $relevant$
```

The main procedure for this version of SHAKE is shown in Alg. 2. Its structure is almost identical to the naive version, but it uses FORMULARELEVANT to determine the relevance of each axiom in the context. A more subtle detail is that SHAKE must revisit all of the axioms in each round, as an axiom's nested quantifiers may be expanded in later rounds. Moreover, the formula state from the goal φ_0 is also part of the main loop. This way the quantifiers in the goal are also lazily expanded.

C. SHAKE *with Frequent Symbols*

Thus far we have used the symbol set abstraction introduced in §IV-A, where we exclude certain basic symbols, such as the built-in SMT-LIB functions, based on the intuition that such prevalent symbols provide little indication of relevance. We now further refine the symbol-set abstraction to reflect this intuition.

Algorithm 2 Refined SHAKE with Quantifiers

```
procedure QUANTIFIERSHAKE($\Gamma = \{\varphi_0, ..., \varphi_n\}$)
    for $\varphi_i \in \Gamma$ do
        # create the formula state
        INITFSTATE($\varphi_i$)
    # assuming $\varphi_0$ is the goal
    $S_{ctx} \leftarrow \varphi_0.S_{visible}$
    $dists, round \leftarrow \{(\varphi_0 : 0)\}, 1$
    repeat
        $acc \leftarrow \emptyset$
        for $\varphi_i \in \Gamma$ do
            $S_{prev} \leftarrow \varphi_i.S_{visible}$
            # possibly expand quantifiers
            if FORMULARELEVANT($\varphi_i, S_{ctx}$) then
                if $\varphi_i \in$ UNREACHED($dists, \Gamma$) then
                    $dists \leftarrow dists \cup \{(\varphi_i : round)\}$
                # update with previous symbols in $\varphi_i$
                $acc \leftarrow acc \cup S_{prev}$
        $S_{ctx} \leftarrow acc \cup S_{ctx}$
        $round \leftarrow round + 1$
    until ISFIXEDPOINT($dists$)
    $max\_dist \leftarrow round + 1$
    for $\varphi_i \in$ UNREACHED($dists, \Gamma$) do
        $dists \leftarrow dists \cup \{(\varphi_i : max\_dist)\}$
    return $dists$
```

In some verification languages, the SMT encoding uses certain symbols pervasively. For example, the function symbol `ApplyTT` is ubiquitous in F* queries. This is expected, as F* is based on dependent types, where terms are proofs, and `ApplyTT` represents term application. However, symbols like `ApplyTT` cause SHAKE to quickly saturate, absorbing many axioms when added to the reached symbol set.

To address this issue, we propose a simple heuristic. We define the frequency of a symbol x to be the ratio of formulas in $\Gamma = \{\varphi_0, ..., \varphi_n\}$ containing x in their symbol set:

$$freq(x) = \frac{|\{\varphi_i \mid \varphi_i \in \Gamma \wedge x \in \text{SYMBOLS}(\varphi_i)\}|}{|\Gamma|}$$

Given a threshold θ, SHAKE excludes all symbols x such that $freq(x) > \theta$, treating them as if they were built-in functions. As a side note, this idea is related to *inverse document frequency* in information retrieval [35]. This simple approach improves pruning on certain F* queries, as we show in the evaluation.

D. SHAKE *with Distance Limit*

SHAKE is similar to *iterative deepening* [36] in spirit. However, SHAKE does not explicitly or implicitly construct a graph. Instead, SHAKE creates "layers" of axioms at different distances. By default, SHAKE runs until a fixed point, dropping axioms that are unreachable at the last layer.

SHAKE's complexity is therefore $O(DN)$, where D is the maximum distance and N is the number of axioms. In practice, our evaluation shows that D is almost always a constant $\leq$

20, while N can be in the thousands, as shown in Figure 1. SHAKE's approach improves efficiency, since a graph-based approach would take $O(N^2)$ time just to construct the graph.

Stopping SHAKE early can also be useful: by setting a distance limit, SHAKE potentially prunes even more irrelevant axioms. However, the other side of the coin is that a shallow distance limit may miss out on relevant axioms that are necessary to the goal.

The choice of distance limit thus appears to present a dilemma. However, we argue that SHAKE can leverage an unsat core as an oracle for nearly-optimal distance: since our main goal is to improve stability, we assume that an initial version of procedure P verifies, and a subsequent version P' may fail due to minor changes. Therefore, we can use the distance limit from the unsat core of P to inform the subsequent runs of P'.

In practice, we envision saving SHAKE's distance limit with source-level annotations. For example, in Dafny, a commonly used attribute is {`:timeLimit N`}, which allows the user to provide a procedure-specific time limit, overriding the default. Related attributes include {`:rlimit N`} and {`:timeLimitMultiplier X`}, which are also solver configurations. Similar annotations also exist in languages like F* and Verus [37].

SHAKE can be configured in a similar way, where the distance value is a procedure attribute. With a fresh procedure (query), the attribute is not present yet, and the solver runs as normal. If verification succeeds, we store the maximum core distance as an attribute. The next time the same procedure is verified, SHAKE uses the stored distance limit and prunes the context accordingly. Small changes in the procedure (e.g., renaming a variable) will have no impact on SHAKE's layering, and the stored limit should still work.

V. EVALUATION

In this section, we evaluate the effectiveness of SHAKE. We describe the experimental setup in §V-A. We show the distribution of distance values produced by SHAKE in §V-B. We then evaluate SHAKE's improvement of context relevance in §V-C and stability in §V-D. We further assess the impact of ignoring frequent symbols in §V-E. Lastly, in §V-F, we evaluate SHAKE's impact on solving performance in terms of run time and number of queries solved.

A. Experimental Setup

In the evaluation, we run SHAKE in two different modes.

- **Default Mode:** SHAKE computes the distances and then prunes the unreachable axioms, i.e., axioms in the last layer discussed in §IV-A.
- **Oracle Mode:** We obtain an "ideal" distance by employing the unsat core as an oracle. We then use SHAKE to prune axioms beyond the oracle distance.

To evaluate stability, we use SHAKE's oracle mode. As discussed in §IV-D, to counter instability, we assume a prior working version of the query that produces a core, from which we obtain the oracle distance.

To evaluate standard solving performance overhead, i.e., without any query mutation, we use the oracle mode along with the default mode. This provides a best-case and worst-case comparison for SHAKE's performance impact as a preprocessor.

By default, SHAKE does not ignore any query-defined symbols based on their frequencies (§IV-C). We only experiment with frequency configuration in §V-E.

We use the default settings for Mariposa [32], including a time limit of 60 seconds for each query. We experiment with recent versions of two SMT solvers, Z3 version 4.12.5 and cvc5 version 1.1.1. We conduct our experiments on machines with an Intel Core i9-9900K (max 5.00 GHz) CPU, 128 GB of RAM, and the Ubuntu 20.04.3 LTS operating system.

B. *Distribution of* SHAKE *Distances*

First, we evaluate how well SHAKE distances reflect the relevance of axioms. Recall that for an original query $\Gamma = \{\varphi_0, ..., \varphi_n\}$, SHAKE computes the distances:

$$dists = \{(\varphi_0 : d_0), ..., (\varphi_n : d_n)\}$$

Let $\Gamma_c \subseteq \Gamma$ be the core provided by the solver. We can then determine the maximum distances for the original query and the core:

$$d_{orig} = \max(d_i \mid (\varphi_i : d_i) \in \Gamma)$$
$$d_{core} = \max(d_i \mid (\varphi_i : d_i) \in \Gamma_c)$$

Intuitively, if $d_{orig} > d_{core}$, then SHAKE is able to differentiate between core and non-core axioms: the more significant the difference is, the more we can safely prune layers in between with no loss of core axioms.

As shown in Figure 5-Figure 9, the maximum distances are upper-bounded by 20 for all queries from the five projects in this study. Moreover, there is usually a clear difference between d_{orig} and d_{core}. As an example, Figure 5 shows the distributions from Komodo$_D$. Note the strong separation between the two: the median d_{core} is 2, while the median d_{orig} is 8. Moreover, the distribution of the d_{core} is light-tailed, where a distance of 3 covers almost the entirety of the query set.

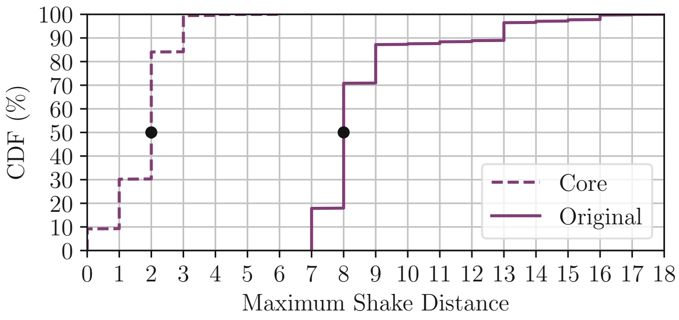


Fig. 5. **Maximum SHAKE Distances for Komodo$_D$.** There is a clear separation between the distance values of core axioms versus original axioms.

However, in Figure 9, we observe that vWasm$_F$ is a bit of an outlier (again). As we discussed in §III-C, the vWasm$_F$

query set starts off with much higher context relevance; thus we do not expect much room for differentiation using SHAKE's distance.

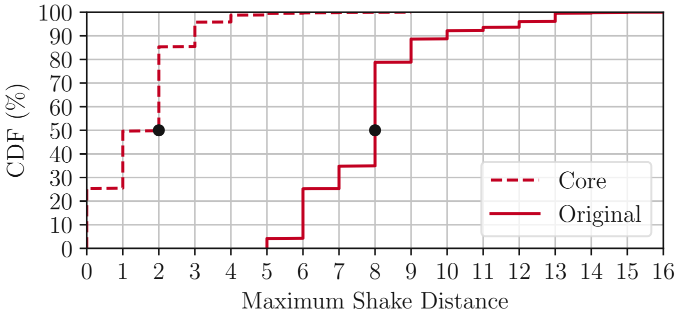


Fig. 6. **Maximum SHAKE Distances for VeriBetrKV$_L$.**

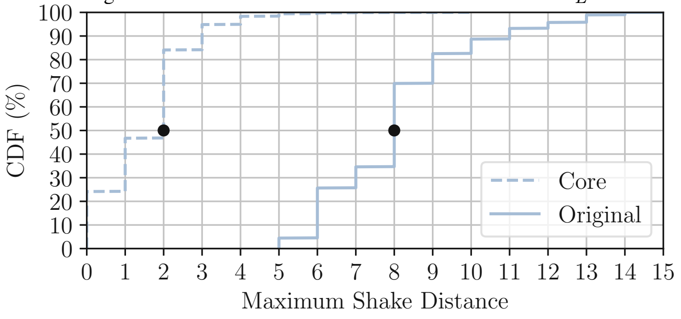


Fig. 7. **Maximum SHAKE Distances for VeriBetrKV$_D$.**

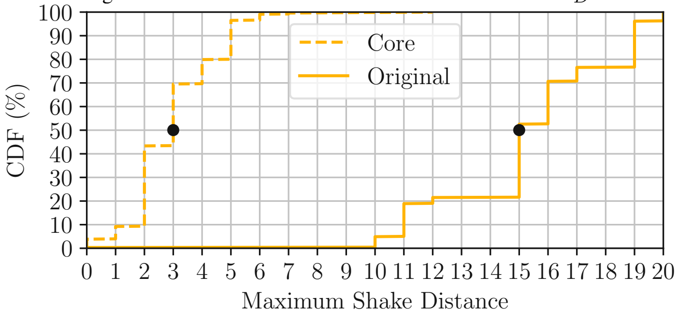


Fig. 8. **Maximum SHAKE Distances for DICE$^\star_F$.**

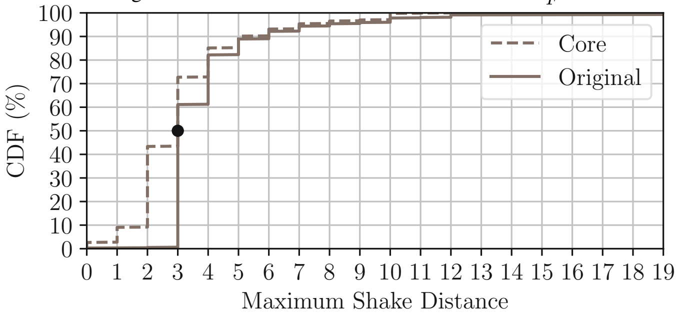


Fig. 9. **Maximum SHAKE Distances for vWasm$_F$.**

C. Context Relevance Ratio

Now that we have demonstrated that SHAKE differentiates core and non-core axioms, we evaluate how much context pruning SHAKE enables. Since our main goal is to mitigate instability, we run SHAKE in oracle mode. As in §III-B, we compute the relevance ratio of the pruned query:

$$\frac{\text{\# core axioms} + 1}{\text{\# axioms after pruning} + 1} \times 100\%$$

In Figure 10, we present the relevance ratios that SHAKE achieves. We see significant improvements over the original queries as shown in Figure 2. For example, in VeriBetrKV$_L$, the median relevance ratio (MRR) is 0.32% in the original queries, while the MRR increases to 3.46% with oracle SHAKE. Overall, SHAKE improves the MRR by 3–10$\times$. We note the intersection on the right side of the plot, where the relevance ratio is 100%. In those cases, SHAKE matches the unsat core when only given the oracle distance.

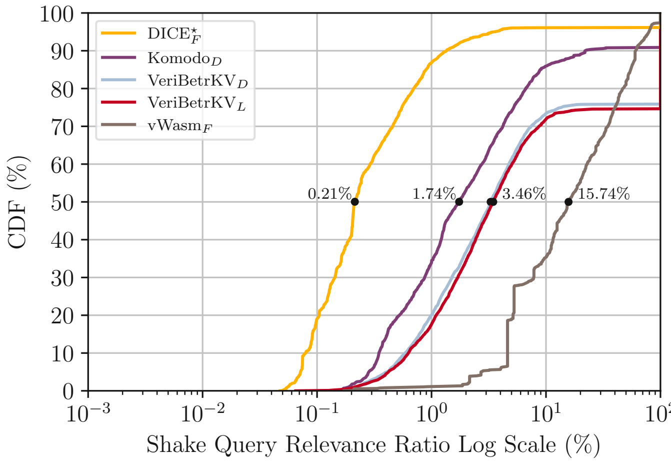


Fig. 10. **Oracle SHAKE Query Context Relevance.** Oracle SHAKE shows improvement of context relevance over Figure 2 by 3–10$\times$.

D. Stability Improvement

Next, we evaluate if the improved context relevance translates into improved stability. We assess stability in the same way as in the unsat core experiments in §III-C, both from a preservation and mitigation perspective.

In Figure 11, we report the stability scores for oracle SHAKE on Z3 version 4.12.5. We include all of the unstable queries found in the original Mariposa query set (just as we did in Figure 3), and then we sample roughly the same number of stable queries (110 from each project). We observe that SHAKE generally preserves stability, and achieves reasonable success mitigating instability, with an overall mitigation score of 29.7%. We also see that the naive SHAKE from §IV-A performs much worse, achieving an overall mitigation score of only 11%.

We observe that DICE$^\star_F$ sees much less mitigation. We attribute this to F*'s pervasive use of certain function symbols (such as `ApplyTT`) in its query encoding. In §V-E, we evaluate the effectiveness of suppressing such symbols based on their frequency. We also observe that SHAKE does not help with the unstable queries in vWasm$_F$. Since the unsat core is not effective on vWasm$_F$, this is unsurprising.

To further validate the stability improvement, we also evaluate SHAKE with cvc5 version 1.1.1. However, cvc5

Project	Original Count		Oracle Naive SHAKE		Oracle SHAKE	
	Stable	Unstable	Preservation	Mitigation	Preservation	Mitigation
Komodo$_D$	110	93	99.1%	7.5%	100.0%	25.8%
VeriBetrKV$_D$	110	172	100.0%	12.2%	98.2%	23.3%
VeriBetrKV$_L$	110	256	100.0%	11.7%	100.0%	37.9%
DICE$^\star_F$	110	20	100.0%	10.0%	100.0%	5.0%
vWasm$_F$	110	4	100.0%	0.0%	96.4%	0.0%
Overall	550	545	99.8%	11.0%	98.9%	29.7%

Fig. 11. **Oracle SHAKE Query Stability on Z3 4.12.5.** We include oracle naive SHAKE (middle column) from §IV-A for comparison. Oracle SHAKE, which employs the quantifier handling strategy from §IV-B, shows similar preservation, but stronger mitigation results.

is known to not work well with queries from Dafny and F*, as acknowledged by cvc5's developers [15]. In fact, to make this evaluation possible, we had to first syntactically transform the queries into a format cvc5 could parse. Even then, cvc5 times out on many of the original queries (whereas Z3 succeeds for nearly all of them). Hence, we only evaluate the stability of original queries that do not timeout with cvc5. This necessarily introduces bias in the resulting query sample, so the stabilization results from Z3 and cvc5 should not be directly compared.

With that caveat in mind, we present the stability scores for oracle SHAKE on cvc5 in Figure 12. Generally, the preservation scores are quite strong. The overall mitigation score of 41.3% is promising as well.

Project	Original Count		Oracle SHAKE	
	Stable	Unstable	Preservation	Mitigation
Komodo$_D$	110	36	100.0%	41.7%
VeriBetrKV$_D$	110	143	94.5%	48.3%
VeriBetrKV$_L$	110	210	100.0%	37.1%
DICE$^\star_F$	110	17	100.0%	100.0%
vWasm$_F$	110	27	99.1%	0.0%
Overall	550	433	98.7%	41.3%

Fig. 12. **Oracle SHAKE Query Stability on cvc5 1.1.1.**

E. Frequency Configuration

As discussed in §IV-C, SHAKE can optionally take in a threshold θ and ignore any symbol x such that $freq(x) > \theta$. We now evaluate if this configuration can help with stability. Intuitively, if θ is set properly, SHAKE can ignore trivial matches due to pervasively used symbols. However, if θ is too low, SHAKE may not reach axioms that are actually relevant, e.g., the ones in the core.

We continue to use the oracle mode for this experiment. Recall that SHAKE assigns the unreachable axioms to the maximum distance. When core axioms end up being unreachable, oracle SHAKE cannot safely prune any axioms, since this could introduce incompleteness. Therefore, in addition to the mean relevance ratio (MRR), we also report the *fallback rate* (FR), which is the percentage of queries where oracle SHAKE cannot prune any axioms.

First, we discuss the choice of θ with an experiment on query relevance. $\theta = 1.00$ means no symbols are pruned based on frequency. In Figure 13, we observe that there is a trade-off between the relevance ratio and the fallback rate. For example, in Komodo$_D$, $\theta = 0.15$ achieves the highest MRR, but also has the highest FR. In vWasm$_F$, since the context starts with high MRR, lower θ values only increase FR. In general, $\theta = 1.00$ (no frequency pruning) tends to balance the two metrics.

		Orig.	$\theta = 1.00$	$\theta = 0.30$	$\theta = 0.15$
Komodo$_D$	MRR	0.57	1.74	1.74	2.40
	FR	–	0.39	6.08	13.14
VeriBetrKV$_D$	MRR	0.33	3.28	3.35	2.51
	FR	–	1.45	5.74	28.49
VeriBetrKV$_L$	MRR	0.32	3.46	3.59	3.03
	FR	–	1.42	5.45	15.91
DICE$^\star_F$	MRR	0.06	0.21	0.32	0.88
	FR	–	4.44	5.90	7.10
vWasm$_F$	MRR	3.76	15.74	16.0	16.22
	FR	–	5.99	6.11	12.51

Fig. 13. **Oracle SHAKE Context Relevance with Frequency Configuration.** Higher MRR means more relevant context. Higher FR means more queries for which oracle SHAKE does not prune any axioms.

However, for DICE$^\star_F$, the results indicate that $\theta = 0.15$ is a promising setting, since the MRR is increased by $4\times$ with respect to $\theta = 1.00$, while sacrificing three percentage points of FR. We test the stability of using $\theta = 0.15$ on DICE$^\star_F$ with Z3 and find that it improves stability by $6\times$ compared to oracle SHAKE with $\theta = 1.00$.

F. Solving Performance Impact

Proof instability is a pernicious problem in program verification, so it might be reasonable to expect developers to be willing to trade worse solving performance for greater stability. Fortunately, our results show that such a trade is largely unnecessary: SHAKE adds relatively little overhead and even improves performance in some cases.

To evaluate solving performance, for each solver (Z3 and cvc5), we compare the following three scenarios.

- **Baseline.** The original queries are directly given to the solver.
- **Default SHAKE.** The queries are preprocessed by SHAKE in default mode and then given to the solver.
- **Oracle SHAKE.** The queries are preprocessed by SHAKE in oracle mode and then given to the solver.

Since SHAKE is a preprocessor, its runtime includes the time spent on computing the distances and the time spent in IO. When reporting the runtime, we exclude the latter, since we expect SHAKE to eventually be incorporated directly into solvers, where parsing is already being done. Therefore, the runtime for the SHAKE modes is the time spent on computing the distances plus the time spent by the solver on the pruned queries. Each query is given a 60 second timeout, so if SHAKE distance computation and solver together takes more than that, the query is not considered solved.

First we present the number of queries solved in each scenario in Figure 14. Generally SHAKE adds a minor overhead to Z3, but sometimes solves a few more in oracle mode. However, if we consider cvc5, SHAKE usually improves the number of queries solved, even in default mode. Notably, in DICE$^{\star}_{F}$, cvc5 solves 259 queries in the baseline; even with default SHAKE, it solves 190 more (+79%); with oracle SHAKE, it solves 424 more (+163%).

	Solver	Baseline	Default	Oracle
Komodo$_D$	Z3 cvc5	1,983 342	-0.10% +1.75%	+0.30% +21.64%
VeriBetrKV$_D$	Z3 cvc5	5,103 2,571	-0.78% +9.14%	-0.61% +20.77%
VeriBetrKV$_L$	Z3 cvc5	5,167 3158	-0.41% +8.90%	-0.04% +13.01%
DICE$^{\star}_{F}$	Z3 cvc5	1,493 259	-0.07% +73.36%	+0.33% +163.71%
vWasm$_F$	Z3 cvc5	1,733 1,630	-0.29% -0.12%	-0.35% -0.12%
Overall	Z3 cvc5	15,479 7,960	-0.45% +8.92%	-0.18% +18.10%

Fig. 14. **Queries Solved with SHAKE as a Preprocessor.**

To present the runtime performance, we use survival plots; Brain et al. [38] provide a detailed explanation, but in short, a survival plot shows the cumulative number of queries solved within a total time budget. Therefore, a curve that is higher and to the left indicates better performance.

In each plot, we show six curves, based on the three scenarios for each of the two solvers. Generally, SHAKE adds a minor overhead to Z3, but often improves the solving speed on cvc5. For example, in Figure 16, we show the survival plot for VeriBetrKV$_D$. SHAKE's impact on Z3 is almost negligible, whether in default or oracle mode. However, for cvc5, SHAKE does improve on the solving speed, as well as the number of queries solved, not only in oracle mode, but also in default mode. In Figure 17, VeriBetrKV$_L$ shows a similar trend as in VeriBetrKV$_D$.

In Figure 18, we show the results for DICE$^{\star}_{F}$. We observe that default SHAKE adds a minor overhead to Z3, but oracle SHAKE has little impact. On cvc5, as we discussed earlier, SHAKE significantly improves the number of queries solved and improves the runtime as well.

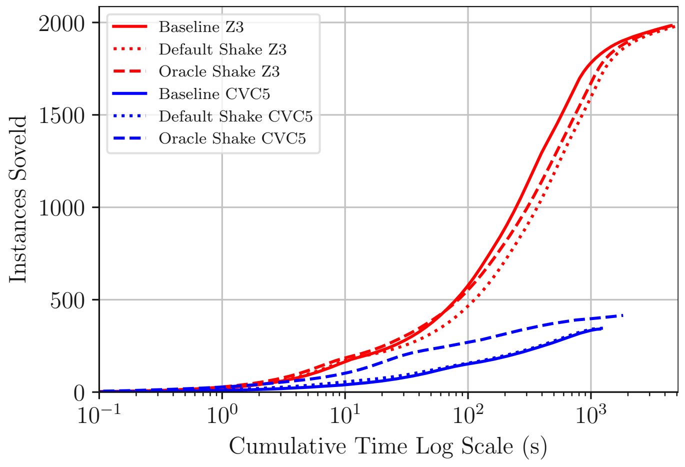


Fig. 15. **SHAKE Performance Survival Plot for Komodo$_D$.**

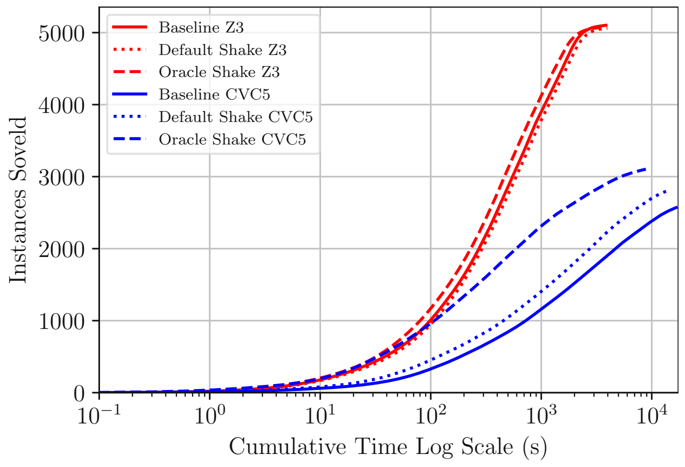


Fig. 16. **SHAKE Performance Survival Plot for VeriBetrKV$_D$.**

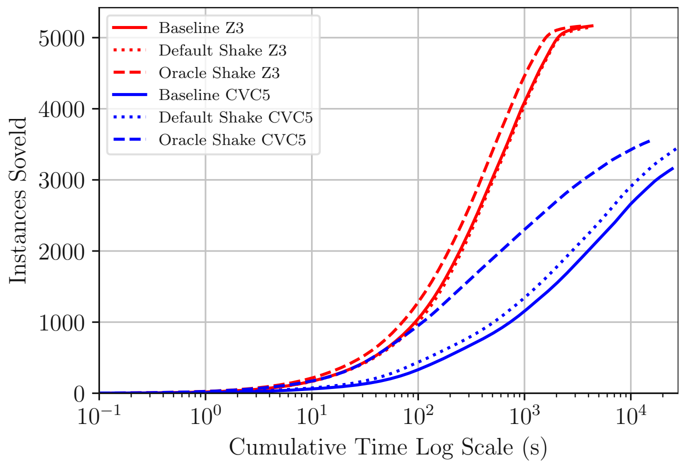


Fig. 17. **SHAKE Performance Survival Plot for VeriBetrKV$_L$.**

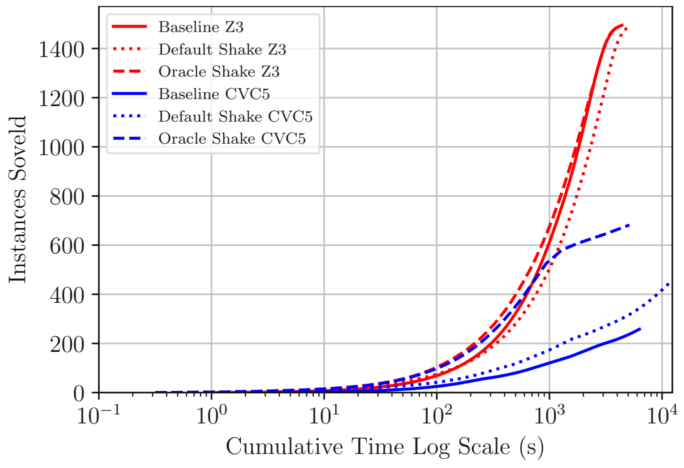


Fig. 18. **Shake Performance Survival Plot for DICE$^{\star}_{F}$.**

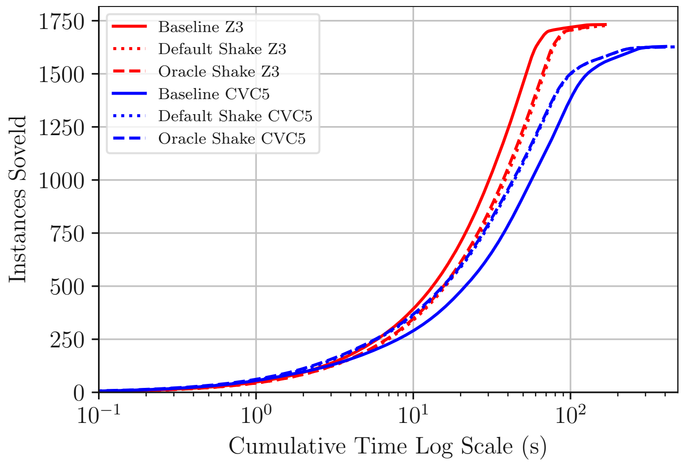


Fig. 19. **Shake Performance Survival Plot for vWasm$_F$.**

VI. Related Work

The problem of proof instability in the context of program verification has been a long standing issue. For example, Hawblitzel et al. bemoan the instability of certain SMT queries [19], and the Komodo authors describe proof instability as "the most frustrating recurring problem" [18].

The Mariposa project [15] is the first effort to quantify in a statistically rigorous way the instability of SMT queries with respect to a solver. The authors measure instability in six large-scale verification projects across eight SMT solver versions. However, their focus is on quantifying instability, rather than understanding or mitigating it.

Our Shake technique resembles an algorithm first implemented in the Sumo INference Engine [39]. The Sine algorithm selects relevant axioms in automated theorem proving (ATP) problems [24]. As in Shake, Sine uses overlapping symbols to iteratively determine relevance. A similar strategy was later employed by the lightweight relevance filtering algorithm [40]. However, these algorithms target ATP problems, e.g., those from TPTP [41], which usually covers domains outside those in the SMT queries produced by program verification. Moreover, a major difference between Shake and these algorithms is the strategy Shake employs to handle quantified expressions. In Shake, we make use of quantifier patterns and perform lazy quantifier expansion, which is not present in the earlier algorithms.

VII. Limitations

This work has several limitations. First, we have only studied verification projects written in Dafny and F*, which do not necessarily represent the entire spectrum of automated program verification. For example, we have excluded Mariposa's Komodo$_S$, since it is restricted to the decidable fragments of SMT and does not fit our description of VCG in §III-D. Second, unsatisfiable cores have guided much of our analysis and experiments, but the solver-produced core is not a perfect oracle of relevant assertions. For example, the solver makes no guarantee about the minimality (necessity) of the core assertions. Third, our proposed technique, Shake, needs to assume an oracle distance limit and/or a frequency threshold to be effective. While the assumption of oracle configurations can be met when dealing with unstable queries, ideally we would like to remove this dependency, possibly by integrating Shake into the SMT solver in future work. Lastly, Shake works at the SMT level, and thus may have less precision compared to VCG-level pruning. Nevertheless, Shake demonstrates the general applicability of context pruning to improve stability, and we leave language-specific adaptations to future work.

VIII. Conclusion

In this work, we empirically study the problem of proof instability in SMT-based program verification. We find that irrelevant context is a major source of instability. We then propose Shake, a novel SMT-level context pruning algorithm as a mitigation technique. We demonstrate that Shake can improve the stability of automated program verification using queries from real-world projects. Furthermore, we show that Shake can potentially improve standard SMT-solving performance on these queries as well. We hope our work offers useful insights into the phenomenon of instability and the connection between automated program verification and theorem proving.

IX. Acknowledgement

Chris Hawblitzel and Doug Woos worked on a prototype algorithm for SMT-level tree-shaking in 2016. This work is a redesign and extension of that effort. We thank Haniel Barbosa and Livia Sun for their advice on cvc5 configuration; Jialin Li for her suggestion of storing distance limit as a procedure attribute; and the anonymous reviewers for their helpful feedback on the paper.

This work was supported in part by the National Science Foundation (NSF) under grant 2224279, funding from AFRL and DARPA under Agreement FA8750-24-9-1000, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab).

References

[1] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., 2010.

[2] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent Types and Multi-Monadic Effects in F*," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[3] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, 1969.

[4] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Commun. ACM*, aug 1975.

[5] C. Barrett, A. Stump, C. Tinelli *et al.*, "The SMT-lib Standard: Version 2.0," in *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2010.

[6] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[7] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2022.

[8] J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel, "Linear Types for Large-Scale Systems Verification," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2022.

[9] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin, "HACLxN: Verified generic SIMD crypto (for all your favorite platforms)," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2020.

[10] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[11] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin, "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[12] Y. Zhou, S. Gibson, S. Cai, M. Winchell, and B. Parno, "Galápagos: Developing verified low-level cryptography on heterogeneous hardware," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2023.

[13] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "EverParse: Verified secure Zero-Copy parsers for authenticated message formats," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019.

[14] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe Multilingual Software Sandboxing using WebAssembly," in *Proceedings of the USENIX Security Symposium*, August 2022.

[15] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, "Mariposa: Measuring SMT instability in automated program verification," in *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, October 2023.

[16] A. Tomb and J.-B. Tristan, "Avoiding verification brittleness in Dafny," https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/, 2023.

[17] M. Dodds, "Formally Verifying Industry Cryptography," *IEEE Security and Privacy Magazine*, vol. 20, no. 3, 2022.

[18] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[19] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad Apps: End-to-End Security via Automated Full-System Verification," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[20] K. R. M. Leino and C. Pit-Claudel, "Trigger Selection Strategies to Stabilize Program Verifiers," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, S. Chaudhuri and A. Farzan, Eds., 2016.

[21] J. W. Cutler, E. Torlak, and M. Hicks, "Improving the stability of type soundness proofs in Dafny," in *Proceedings of the First Workshop on Dafny*, 2024.

[22] S. Ho and C. Pit-Claudel, "Incremental proof development in Dafny with module-based induction," in *Proceedings of the First Workshop on Dafny*, 2024.

[23] S. McLaughlin, G.-A. Jaloyan, T. Xiang, and F. Rabe, "Enhancing proof stability," in *Proceedings of the First Workshop on Dafny*, 2024.

[24] M. Fitting, *First-order Logic and Automated Theorem Proving*. Springer Science & Business Media, 2012.

[25] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: Proving Practical Distributed Systems Correct," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[26] A. Arasu, T. Ramananandro, A. Rastogi, N. Swamy, A. Fromherz, K. Hietala, B. Parno, and R. Ramamurthy, "FastVer2: A provably correct monitor for concurrent, key-value stores," in *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)*, January 2023.

[27] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, "Storage Systems are Distributed Systems (So Verify Them That Way!)," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[28] T. Hance, Y. Zhou, A. Lattuada, R. Achermann, A. Conway, R. Stutsman, G. Zellweger, C. Hawblitzel, J. Howell, and B. Parno, "Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2023.

[29] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran *et al.*, "Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[30] B. Cook, "Formal reasoning about the security of amazon web services," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 38–47.

[31] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta, "Hardening Attack Surfaces with Formally Proven Binary Format Parsers," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2022. [Online]. Available: https://www.fstar-lang.org/papers/EverParse3D.pdf

[32] "Mariposa Public Repository," https://github.com/secure-foundations/mariposa, accessed: May 2023.

[33] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1980, aAI8011683.

[34] M. Moskal, "Programming with triggers," in *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2009.

[35] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[36] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[37] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2023.

[38] M. Brain, J. H. Davenport, and A. Griggio, "Benchmarking solvers, SAT-style." in *SC^2@ ISSAC*, 2017.

[39] K. Hoder and A. Voronkov, "Sine qua non for large theory reasoning," in *International Conference on Automated Deduction*. Springer, 2011, pp. 299–314.

[40] J. Meng and L. C. Paulson, "Lightweight Relevance Filtering for Machine-Generated Resolution Problems," *Journal of Applied Logic*, vol. 7, no. 1, pp. 41–57, 2009.

[41] G. Sutcliffe and C. Suttner, "The TPTP Problem Library," *Journal of Automated Reasoning*, vol. 21, pp. 177–203, 1998.

Formal Methods in Computer-Aided Design 2024

Easter Egg: Equality Reasoning Based on E-Graphs with Multiple Assumptions

Eytan Singher and Shachar Itzhaky
Technion - Israel Institute of Technology, Haifa, Israel
{eytan.s,shachari}@cs.technion.ac.il

***Abstract*—E-graphs are a prominent data structure that has been increasing in popularity in recent years due to their expanding range of applications in various formal reasoning tasks. E-graphs allow systematic and efficient treatment of equality, which is pervasive in automated reasoning based on proofs.**

E-graphs handle equality well, but are severely limited in their handling of case splitting and other aspects of propositional reasoning, such as resolution, which introduce branching in provers and solvers. As a consequence, most tools resort to using e-graphs locally, recreating them ad-hoc when they are needed, and then discarding them. In exploratory scenarios, where it is necessary to retain multiple branches simultaneously, this limitation proves to be prohibitive. In particular, in theory exploration—a process where lemmas are discovered and then proven—this poses a significant challenge. Theory exploration must enumerate a space of possible assumptions, and must retain all of them to make progress. This poses a severe limitation on the ability to harness e-graphs for the task.

Our key observation is that in exploratory reasoning tasks, branching represents versions of the same e-graph each with an added assumption, such as "$x > y$" or "is_sorted l". Essentially, each e-graph represents an equality relation, and each branch corresponds to a matching coarsened equality relation. Based on this observation, we present an extension to e-graphs, called *Colored E-Graphs*, as a way to efficiently represent all of the coarsened equality relations in a single structure. A colored e-graph is a memory-efficient equivalent of multiple copies of an e-graph, with a much lower overhead. This is attained by sharing as much as possible between different cases, while carefully tracking which conclusion is true under which assumption. It can be viewed as adding multiple "color-coded" layers on top of the original e-graph structure, representing different assumptions.

We run experiments and demonstrate that our colored e-graphs can support large numbers of assumptions and terms with space requirements that are about $10\times$ lower, and with slightly improved performance.

I. Introduction

E-graphs are a versatile data structure that is used for various tasks of automated reasoning, including theorem proving and synthesis. E-graphs have been popularized in compiler optimizations thanks to their ability to support efficient *rewrites* over a large set of terms, while keeping a compact representation of all possible rewrite outcomes. This mechanism is known as *equality saturation*. It provides a powerful engine that allows a reasoner to generate all equality consequences of a set of known, universally quantified, equalities. Possible uses include selecting the best equivalent of an expression according to some desired metric, such as run-time efficiency [29], size [10], [22], or precision [23] (when used as a compilation phase) and a generalized form of unification, called e-unification, for application of inference steps (when used for proof search).

In this work we focus on a stepping stone for what we address as *exploratory reasoning*: a range of tasks including all the above optimization procedures, as well as theory exploration [26], rewrite rule inference [20], and proof search [16], [5], [14]. Exploratory reasoning, in general, can be thought of as any reasoning task navigating a large space of potential goals or sub-goals that need to be selected based on some criteria. Our motivating example comes from TheSy and Ruler, both of which are theory exploration systems based on e-graphs. A theory exploration system attempts to both discover and prove mathematical properties from a set of definitions and known lemmas. Most of the difficulty in theory exploration comes from the generation and filtering of candidates, rather then from the proof procedure itself. TheSy does so by efficiently filtering a large set of potential conjectures using e-graphs for equality reasoning, and evaluating which should be potentially proved. While e-graphs are effective for equality reasoning [30], handling branching, such as case splitting during proof search, do not have a common solution, and are treated ad-hoc. For example, a special type of node is introduced in [29] to deal with loop conditions, while in [7] a special operator is introduced to reason on expressions under certain contexts, and [26] creates full copies of the e-graph for each branch being explored.

To illustrate this difficulty, we zoom in on an example from theory exploration. As an example scenario, consider trying to discover and prove lemmas on sorted lists: a library containing functions find, is_sorted, and bin_search. We expect to discover lemmas involving these functions; one such lemma might be the property: $\mathsf{is\_sorted}\,l \rightarrow \mathsf{bin\_search}\,l\,v = \mathsf{find}\,l\,v$. State-of-the-art theory exploration systems [12], [20], [26] have some enumeration strategy over expressions in order to discover candidates. A challenge presents itself when some lemmas in the space require an assumption, in this case $\mathsf{is\_sorted}\,l$. When dealing with e-graphs, adding an assumption would *globally* affect all terms involved in the enumeration, making it impossible to separate conclusions stemming from different assumptions. Because the system cannot know in advance which assumptions will become relevant for discovering equalities, it is required that it also generate and test multiple candidate assumptions. An immediate solution is to create one copy of the graph per assumption, but doing so can significantly increase the memory usage. Moreover, lemmas may depend on

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_13

one another; for example, $\mathsf{is\_sorted}\, l \to \mathsf{bin\_search}\, l\, v = \mathsf{find}\, l\, v$ depends on transitivity of $\leq$ $(x \leq y \wedge y \leq z \to x \leq z)$. Therefore, just trying the candidates one at a time would mean that the system would prematurely discard candidates depending on the order in which they are tested; alternatively, for each candidate that is validated and becomes a lemma, it would be forced to re-try all the previously failed attempts, which is highly costly.

To overcome this difficulty, we propose an extension of the e-graph data structure. An e-graph naturally represents a congruence relation $\cong$, which is an equality relation over terms (with function applications), which maintains $x \cong y \vdash f(x) \cong f(y)$. The congruence relation is maintained in the e-graph as a set of equivalence classes (e-classes), which can be merged as part of updating the underlying relation. We extend the e-graph data structure into a *Colored E-Graph* to maintain multiple congruence relations at once, where each relation is associated with a color. Our key observation is that each added assumption, can be treated as a new congruence relation, but is only a coarsening of the original relation. The coarsening, then, can be represented as a set of additional merges of e-classes on top of the original e-graph. The main benefit is reducing memory consumption by re-using and sharing most of the e-classes between colors. Going back to the sorted list example, in the colored e-graph there will be a red relation for assuming $x \leq y \wedge y \leq z$, and a blue relation for assuming $\mathsf{is\_sorted}\, l$. Thanks to the size reduction, multiple relations can exist at once, and thus the lemma $\mathsf{is\_sorted}\, l \to \mathsf{bin\_search}\, l\, v = \mathsf{find}\, l\, v$ can be discovered after transitivity of $\leq$ is proven, but without dependency on the order of exploration. Colored e-graphs also support having a hierarchy between different colors, which can benefit from additional sharing of e-classes. For example, the red color representing $x \leq y \wedge y \leq z$ is itself a coarsening of some green color representing just the assumption $x \leq y$.

While the memory footprint for each color is smaller, maintaining the congruence relation and the data structure invariants becomes more challenging. To address this we present specialized data-structure modifications and evaluate them. First, we set up a multi-level union-find where the lowest level corresponds to the root congruence. Second, we change how congruence closure is applied to the individual congruence relations while taking advantage of the sharing between each such relation and the root. Lastly, we present a technique for efficient e-matching over all the relations at once.

Our contributions are:

1) The observation that assumptions induce coarsened e-graphs that share much of the original structure.
2) Algorithms for colored e-graphs operations.
3) Optimizations on top of the basic algorithms to significantly improve resource usage.
4) A colored e-graph implementation, *Easter Egg*[1] and an evaluation that shows an improvement factor in memory usage over the existing baseline, while maintaining similar run-time performance.

[1]https://github.com/eytans/egg/tree/features/color_splits

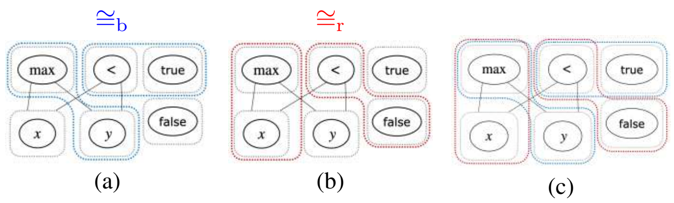


Fig. 1. Example e-graph with two colored layers; (a) is blue, (b) is red, (c) shows them combined.

II. Overview

From this point we assume familiarity with the basic e-graph structure which includes a union-find, hashcons, and an e-class map, as well as the basic operations of add, merge, rebuild, and e-matching (and consequently rewriting). For readers unfamiliar with e-graphs, or with deferred rebuilding, which was introduced in [30], additional background is given in Appendix A.

Colored E-graphs are an extension of e-graphs devised to add a generic approach for supporting conditional reasoning to e-graphs. Existing exploratory reasoning systems such as TheSy [26] and Ruler [20] utilize equality saturation with e-graphs for discovering new rewrite rules, but are limited in the presence of conditionals. For example, let $t := \mathsf{max}(x, y)$, then reasoning about the cases $x < y$ and $x \geq y$ separately is desirable: in the first case $t \cong x$, and in the second $t \cong y$. Without any assumptions, we can say neither and rewriting of t is blocked. The approach in [26] involves a prover that creates an e-graph *clone* for each case in case splitting, such as for $x < y$ and $x \geq y$. This process, however, incurs high runtime and memory costs. Non-relevant terms in the e-graph are unnecessarily duplicated, and rewrites are redundantly applied to these copies. Further case splits compound this issue, leading to an exponential increase in the number of clones with additional nested splits.

Colored e-graphs are designed to avoid duplication via sharing of the common terms, thus storing them only once when possible. The e-graph structure becomes *layered*: the lowermost layer represents a congruence relation over terms that is true in all cases (represented, normally, as e-classes containing e-nodes). On top of it are layered additional congruence relations that arise from various assumptions.

Going back to our example, the corresponding e-graph is shown in Figure 1, containing the terms $\mathsf{max}(x, y)$, $x < y$, true and false. Layers corresponding to assumptions $x < y$ and $x \geq y$ are shown in 1(a) and 1(b). To evoke intuition, we associate with each layer a unique *color*, and paint their e-classes (dotted outlines, in depicted e-graphs) accordingly. Conventionally, the lowermost layer is associated with the color black. In the subsequent example we will use blue for $x < y$ and red for $x \geq y$ when referring to the example. In the blue layer, $(x < y) \cong_b \mathsf{true}$ and $\mathsf{max}(x, y) \cong_b y$; in the red layer, $(x < y) \cong_r \mathsf{false}$ and $\mathsf{max}(x, y) \cong_r x$. This is shown via the corresponding blue and red dotted borders. Figure 1(c) shows a depiction where both colors are overlain on the same graph, which is a more faithful representation of the concept of colored e-graphs, although this visualization is clearly not scalable to larger graphs. In Figure 2, a larger graph can be seen that includes the terms $\mathsf{max}(x, y) - \mathsf{min}(x, y)$ and $|x - y|$. An overlain graph will be quite incomprehensible in this case, so the layers are shown separately; it can be easily discerned that $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong_{\mathrm{b}} |x - y|$ as well as $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong_{\mathrm{r}} |x - y|$.

Both additional layers, blue and red, use existing (black) e-nodes, with each color represented by further unions of e-classes in the black congruence relation. Each color's congruence $\cong_c$ is a *coarsening* of the black congruence, $\cong$, as $\cong \subseteq \cong_c$. In complex cases like the generalization of $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong |x - y|$ to $\mathsf{max}(x, y, z) - \mathsf{min}(x, y, z) \cong \mathsf{max}(|x - y|, |x - z|, |y - z|)$, the colored e-graphs have an important layered structure. This scenario requires reasoning about additional assumptions, building additional layers, such as $x < y \land y < z$ on top of $x < y$ (and respectively $x \geq y \land y < z$ on top of $x \geq y$). These additional layers will reuse the blue and red ones, as they are a coarsening of the respective $\cong_{\mathrm{b}}$ and $\cong_{\mathrm{r}}$.

Before diving into the design of colored e-graphs, it is better to start with their expected semantics. One way to understand the semantics of colored e-graphs is by analogy to a set of clones, i.e. separate e-graphs $\mathcal{E}$. One e-graph represents the base congruence $\cong$, and one e-graph per color c represents $\cong_c$. All e-graphs in $\mathcal{E}$ conceptually represent the same terms partitioned differently into e-classes. Thus, they have the same e-nodes, except that the choice of e-class id (the representative) may be different according to the composition of the e-classes. We will call the e-classes of the color congruences *colored e-classes*. A union in any layer, black or colored, is in effect a union applied to the respective e-graph and all its descendants. Thus, a union in the black layer (i.e. the original e-graph) is analogous to a union in *all* of the e-graphs of the corresponding e-classes; this maintains the invariant that every colored e-class is a union of (one or more) black e-classes. The colored e-graph semantics of the other operations—insertion, congruence closure, and e-matching—are the same as if they were performed across all clones.

A guiding observation in the design is that in equality saturation based exploratory reasoning tasks, where the e-graphs are extensive, each assumption leads to modest increase in congruences. Colored e-graphs are adapted to this scenario. The basic presupposition is that most colored layers, like the blue layer in Figure 2, do not involve an excessive amount of additional unions. In these cases, the space savings from not duplicating black e-nodes more than compensate for the added complexity in managing colored e-classes. With careful tweaks and a few optimizations, we show that we improve upon a clone-based approach. Importantly, if the assumption leads to an inordinate increase in additional unions, the clone-based approach could be more appropriate, and it is possible to use a clone for that specific assumption.

For presentation purposes, we start with a basic implementation that is not very efficient but is effective for understanding the concepts and data structures; then, we indicate some pain points, and move on to describe optimization steps that can alleviate them.

In the basic implementation, all e-nodes reside in the "black" layer, represented by a "vanilla" e-graph implemented in egg, with normal operations. The colored congruences do not have designated e-graphs of their own, and instead, the operations of merge, rebuild, and e-matching have *colored variants*, parameterized by an additional color c, that are semantically analogous to the same operations having been applied, in clone semantics, to the e-graph associated with color c in $\mathcal{E}$. (Insertion is deferred to later.)

Colored merge. In colored e-graphs, the union-find structure used for merging, which traditionally holds all e-class ids, is optimized. A master copy retains black unions, while each color layer has a *smaller* union-find for merged representative e-classes of the parent layer. This approach avoids replication of data across layers.

Colored e-matching. The e-class map is only saved for the black layer. This is sufficient, because an e-class in color c is always going to be a union of black e-classes, and all that is required for e-matching is finding e-nodes with a particular root (operator) in the course of the top-down traversal. So the union can be searched on demand by collecting all the "c-color siblings" of the e-class and searching them as well.

Colored congruence closure. In egg, the e-graph maintains congruence by cycling through a work list of altered classes, re-canonizing their parents, and identifying unions to complete congruence through duplicate detection. In colored e-graphs the root will behave the same, but for colored layers there is no single e-class, as the colored e-classes are a equality class of concrete e-classes. For each color, we maintain an additional work list and collect concrete parents from e-classes on demand. This results in a rebuild algorithm similar to egg's, but without updating the hashcons in colored layers, as they are not present.

For a more concrete example, we give a detailed walk-through of equality saturation in a colored e-graph of the red case from Figure 2(b), and show the steps taken to construct this colored layer in Appendix C.

When using the above operations in the context of equality saturation, e-matching is applied for all colors to discover matches for the left-hand sides of rules. For each match, the right-hand side of the rule needs to be inserted into the e-graph and merged or color-merged with the left-hand side. Inserting the e-nodes to the e-graphs makes them available to all layers. This aspect is sound, since we assume that the mere *existence* of a term in an e-graph does not in itself have the semantics of a judgement—it is only the placing e-nodes in the same e-class that asserts an equality. However, in the presence of many colors, and thus many colored matches, the result would be a large volume of e-nodes that are in black e-classes of size 1, as they were created to serve a single color. As

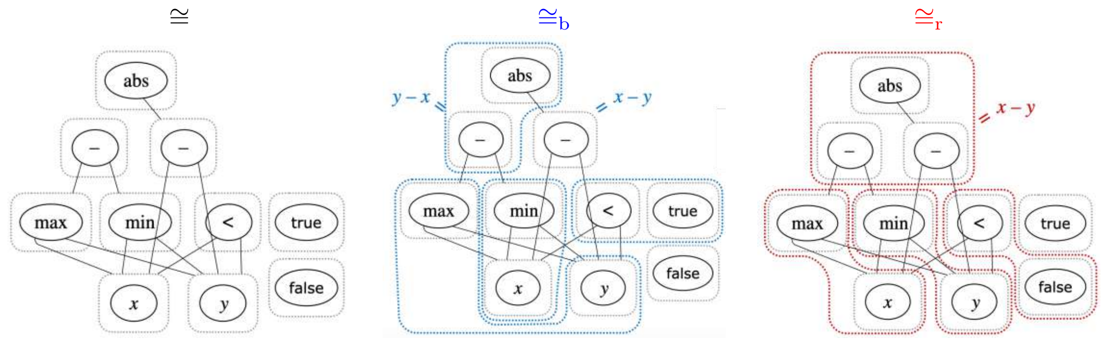


Fig. 2. Proof of $\mathsf{max}(x, y) - \mathsf{min}(x, y) = |x - y|$. The e-nodes corresponding to the two terms are in the same e-class both in the blue layer (b) and in the red (c). It is important to note that the layers are overlain, and that the black nodes are shared; they are separated here for ease of perception.

opposed to a, standard, single e-graph where merging e-classes shrinks the space of e-nodes (because non-equal e-nodes may become equal as a result of canonization), in colored unions it is required that the e-graph maintain both original e-classes, thus losing this advantage. This can put a growing pressure on subsequent e-matching and rebuild operations *in all colors*. Optimizations to improve colored e-graphs, and to address this issue, are presented in section IV.

III. Functional Description

We now introduce some notations and definitions that formalize the description of the e-graph presented in section II. We assume a term language L where terms are constructed using *function symbols*, each with its designated arity. We use $f^{(r)} \in \Sigma[L]$ to say that f is in the *signature* of L and has arity r. A term is then a *tree* whose nodes are labeled by function symbols and a node labeled by f has r children. (In particular, the leaves of a term have nullary function symbols.) Additionally we use the following definitions:

e-class ids	E
e-nodes	$N = \{f(e_1, .., e_r) \mid f^r \in \Sigma, e_i \in E\}$
union-find	$\equiv_{\text{id}} \subseteq E \times E$, $\equiv_{\text{id}}$ is an equivalence relation
e-class map	$M : E \to \mathcal{P}(N)$
parent map	$P = \{e \mapsto \{(n, e') \mid e' \in E \wedge n \in M(e') \wedge n = f(\ldots, e, \ldots)\} \mid e \in E\}$
hashcons	$H = \{n \mapsto e \mid n \in M(e)\}$

Semantically, every e-class represents a set of terms over Σ. We will use the notation $[t]$ to refer to e-class id of the equality class that represents (among other terms), the term t.

The union-find structure offers an operation, $\mathit{find}(e)$, that returns a unique representative id of the equivalence class (of $\equiv_{\text{id}}$) that contains e. That is, $\mathit{find}(e) \equiv_{\text{id}} e$ and for all $e_1 \equiv_{\text{id}} e_2$, $\mathit{find}(e_1) = \mathit{find}(e_2)$.

On top of these basic structures, we introduce a set of *colors*. As explained in section II, colors are organized in a tree whose root is the initial color ("black"). We mark the root color $\varnothing$ and assign to every non-root color c a *parent color* $p(c)$.

colors	$C = \{\varnothing, \ldots\}$
parent colors	$p : C \setminus \{\varnothing\} \to C$

The colored e-graph will now hold multiple union-find structures, one per color. They define a family of equivalence relations $\equiv_c$ by induction on the path from $\varnothing$ to c.

- ▷ $\equiv_\varnothing = \equiv_{\text{id}}$; $\mathit{find}_\varnothing(e) = \mathit{find}(e)$
- ▷ $\equiv_c \subseteq E_{p(c)} \times E_{p(c)}$, where $E_{p(c)} = \{\mathit{find}_{p(c)}(e) \mid e \in E\}$ is the set of all representatives from $\equiv_{p(c)}$. $\mathit{find}_c(e)$ for $e \in E_{p(c)}$ returns a unique identifier in the normal manner of union-find, *i.e.*, $\mathit{find}_c(e) \equiv_c e$ and for all $e_1 \equiv_c e_2$, $\mathit{find}_c(e_1) = \mathit{find}_c(e_2)$.

The definitions over $E_{p(c)}$ are naturally extended to E by (recursive) application of find; i.e., $\mathit{find}_c(e) = \mathit{find}_c(\mathit{find}_{p(c)}(e))$ and $e_1 \equiv_c e_2 \Leftrightarrow \mathit{find}_{p(c)}(e_1) \equiv_c \mathit{find}_{p(c)}(e_2)$. Thus it holds, by construction, that $\equiv_c \supseteq \equiv_{p(c)}$.

The colored e-graph also supports a $\mathit{merge}_c(e_1, e_2)$ operation for each color c where $e_1, e_2 \in E_c$. The merge operation may break the congruence relation invariants for c and all its descendants, and thus needs to be fixed. The merged classes are added to $\mathit{worklist}(c')$ for all c' where c' is c or one of its descendant. In egg [30], the invariants are restored periodically by performing a REBUILD pass. To accommodate the colors, we adjust the REBUILD logic to a multi-congruence-relation setting, so that it restores a congruence closure for each color during REBUILD. The main difference is that for a colored congruence relation, the procedure will collect the parents of a colored e-class by combining the sets of parents of all the (root) e-classes contained therein.

Another important colored e-graph operation is e-matching. Colored e-matching is a modification of the e-matching abstract machine presented in [19]. E-matching is performed by an abstract machine M which consists of a program counter, array of registers reg, and backtracking stack bs, in combination with a sequence of instructions that represents a pattern p. The machine will run instructions by order, where each may either fail if its assertion is not met, or produce a set of continuation states. If a continuation state is produced, the machine selects the first one and adds the current instruction to the stack. If no continuation state is produced, the machine backtracks, retrieving the most recent state from the stack and attempting the next available continuation.

To better present our modifications in colored egg, we first shortly introduce some of the original instruction types:

- ▷ bind(in, f, out) — Matches any e-node of the form $f(x_1, \ldots, x_n)$ that resides in the e-class saved in $reg[in]$, storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.
- ▷ compare(i, j) — Asserts $reg[i] == reg[j]$.
- ▷ check$(i, term)$ — Asserts that the e-class $reg[i]$ represents $term$.
- ▷ continue(f, out) — Match any e-node $f(x_1, \ldots, x_n)$ (in *any* e-class), storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.
- ▷ join$(in, reverse_path, out)$ — Match any e-node $f(x_1, \ldots, x_n)$ that is reachable through $reverse\_path$ from the e-class $reg[in]$, storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.

To facilitate matching across various congruence relations, we adjust the machine M to include the, currently being e-matched, colored assumption $color$ in its state. Adapting to $color$ involves changes in compilation and instructions. The two primary scenarios impacted are: during compare(i, j), ensuring $reg[i] \equiv_{color} reg[j]$, and in function application matching represented by a bind instruction. Before each 'bind' instruction, the modified compilation will insert a new 'colored_jump' instruction to try matching the full colored equality class, one "root" e-class at a time. This is achieved by having 'colored_jump(i)' yield all the "colored siblings" of $reg[i]$ in the current $color$, replacing $reg[i]$ with the result. The instruction 'check' can be likewise adjusted, but we point out that, in fact, it can be implemented as a sequence of 'bind's (with respective interleaved 'colored_jump's).

Multipatterns, supported by the abstract machine, enable e-matching against patterns with shared variables, useful for matching the precondition in conditional rewrite rules. This is achieved using the 'continue' instruction, which selects a new root for subsequent sub-patterns. In the colored setting, while 'continue' remains as is, for performance, it's sometimes substituted with 'join'. This alternative instruction also picks a new root, but restricts selection to e-nodes that can reach a specified e-class, linked to a previously matched hole, through child edges in the e-graph. A *reverse path* is provided to further restrict the upward search needed to find such e-nodes. We do not go too deep into the details, but its colored variant will invoke a colored_jump at every level. We point out that egg does not currently implement 'join', and our colored egg supports a special (though frequent) case in which $reverse\_path$ is empty.

The algorithms described here are presented in more depth in Appendix B.

IV. Optimizations

Both rebuilding and e-matching in colored e-graph, as discussed in section II, can be significantly slower compared to a separate, minimized e-graph.

In the rebuilding aspect, two main burdens are that the colored e-graph contains additional e-nodes compared to each of the separate ones, and that building a colored hash-cons (which will be presented shortly) requires going over all the e-classes.

In the e-matching aspect, colored e-matching may produce duplicate results due to the e-graph not being minimized according to the color's congruence relation; that is, colored-congruent terms are not always merged under a single e-class id. To illustrate this, consider a simple e-graph representing the terms $1 \cdot 1$, $1 \cdot x$, $1 \cdot y$, and $x \cdot y$. Introduce a color, blue, where $x \cong_b y$. A simple pattern such as $1 \cdot ?v$ would have three matches, with assignments $?v \mapsto 1, ?v \mapsto x, ?v \mapsto y$. If the blue layer were a separate e-graph, x and y would have been in the same e-class, so one of the matches here is redundant (as far as the blue layer is concerned). Of course, in the black layer they are different matches; the point is, that many terms are added to the graph only as a result of a colored match, so matching them in the black e-graph is mostly useless to the reasoner. On the other hand, their *presence* in the black layer means they cannot ever be merged, leading to duplicate matches, as seen above, even in the respective colored layer(s).

Moreover, when inserting e-nodes to the e-graph, the hash-cons is used to prevent duplication, relying on it being canonized. Adding an e-node from a colored conclusion (following a match modulo $\cong_b$) does not benefit from canonization. In fact, each e-node $f(x_1, \ldots, x_n)$ has a multitude of black representatives that are $\cong_b$-equivalent. Each child x_i in the e-node can be presented by any black id such that $e \in [x_i]_b$, so there are $\prod_i |[x_i]_b|$ representations. These variants are distinct in the root color, so they cannot be de-duplicated as usual.

To address these issues, we present a series of optimizations to the colored e-graph data-structure and the procedures. These optimizations aim to reuse the "root" and ancestor layers as much as possible, both in terms of memory usage and compute. Thus, we can achieve a memory efficient, but effective colored e-graph.

A. Data-structure optimizations

Colored e-nodes. In the basic implementation outlined in section II, adding e-nodes from colored e-matches to the root e-graph may make it very large and increase the cost of all subsequent actions. The optimized version addresses this by introducing *colored e-nodes*, where e-nodes resulting from colored matches are tagged with their inducing colors. Each colored layer has its own colored hash-cons and e-class map, designed to store only the differences from the parent layer, thereby maximizing reuse. The new mappings added are:

e-class color	$EC : E \to C$
colored parent	$P_c = \{(n, e) \mid (n, e) \in P \land EC(e) = c\}$
colored hashcons	$H_c = \{n \mapsto e \mid n \in M(e) \land EC(e) = c\}$

Note that base parents and hashcons from the non-optimized version are incorporated as $P_\varnothing$ and $H_\varnothing$ in colored mappings.

This optimization applies the hierarchy in all operations. For example, while inserting an e-node to a color c, it is looked up in the colored hashcons for c and all its ancestors, $p^*(c)$, and finally, if no match is found, it is inserted into a new e-class e, setting $EC(e) = c$. The colored hashcons H_c is canonized to color c, ensuring that new e-nodes are unique to this layer and avoiding colored duplicates. (Some duplication related to c may still occur in ancestor layers, as their e-nodes are not canonized to c.) The optimization significantly impacts e-matching: previously when matching a function application f, all f-e-nodes in N were considered; now, only those e-nodes n in the colors hierarchy, that is, those satisfying $\exists e.\ n \in M(e) \wedge EC(e) \in p^*(c)$, are examined.

Pruning. Recall that having a coarsening relation between the colors in the hierarchy means that any result found in an ancestor color is also true for the descendant(s). And so, following merges, some of the colored e-nodes could become subsumed by e-nodes that already exist in an ancestor layer. We present an efficient deferred pruning method to remove the redundant e-nodes.

Normal e-graph minimization relies on having all e-nodes canonized. A colored e-graph usually does not canonize all e-nodes to a specific color c (except for $\varnothing$). Rather, H_c contains only the difference from previous layers. To find redundant e-nodes, the colored e-graph builds a transient hashcons during rebuild from all relevant e-nodes that are not c-colored. The new hashcons, H'_c, is created as follows:

$$H'_c = \{\mathit{canonize}_c(n) \mapsto \mathit{find}_c(e) \mid \\ n \in M(e), EC(e) \in p^+(c)\}$$

A c-colored class e can be reduced by removing all e-nodes that already exist in H'_c. While pruning is promising, one must take care that pruned e-nodes are not immediately re-added.

Colored minimization. Another improvement is having multiple colored e-nodes (of the same color) in a single (black) e-class. As mentioned previously, any e-node that resulted from a colored insert had to be in their own e-classes, as no black unions would be performed on them. But, given that $e \equiv_c e' \wedge EC(e) = EC(e') = c$, then the two black e-classes e, e' can be merged as both contain colored e-nodes of the same color and are in the same colored e-class (of the same color). Thus an invariant is kept that each colored equality class has at most one black e-class containing colored e-nodes.

B. Procedure optimizations

Rebuild. When rebuilding, we first reconstruct the congruence relation of the "root" layer. Even though a color, for example blue, will need to rebuild its own congruence, it still holds that $\cong \subseteq \cong_{\mathrm{b}}$. So, any union induced by $\cong$ can be applied to the blue relation. To understand the implications, consider the e-graph representing the terms x, y, $f(x)$, $f(y)$, $f(f(x))$, and $f(g(y))$ where the blue color contains the additional assumption that $g(y) \cong_{\mathrm{b}} f(y)$. If we union x and y, the black congruence will include $f(x) \cong f(y)$ which also holds in the blue relation. But, the rebuilding of the blue congruence invariant will include an additional, deeper (in terms of rebuilding rounds), conclusion $f(f(x)) \cong_{\mathrm{b}} f(g(y))$. This demonstrates how reusing parent relations is useful; the rebuild depth can be reduced by first rebuilding finer relations.

E-match. In e-matching, we implement an optimization where findings on the root layer are also valid for higher layers. To avoid redundant pattern matching, e-matching begins only from $\varnothing$, adding colored assumptions as needed. There are two scenarios for introducing a colored assumption: The first during *compare*(i, j), if $\mathit{reg}[i] \not\equiv_{\mathit{color}} \mathit{reg}[j]$, we explore descendant colors c where $\mathit{reg}[i] \equiv_c \mathit{reg}[j]$, adding states with $\mathit{color} \leftarrow c$ to the backtracking stack bs. The second is on-demand coloring in colored_jump, where jumps to any color c are enabled if $M.\mathit{color} \in p^+(c)$ and the target e-class is otherwise unreachable. We minimize the set of new assumptions to prevent redundant colors. During the updated compare, compare', if a color c is sufficient, its descendants are not added to bs. For to updated colored_jump, colored_jump', e-classes are matched only with their topmost (closest to root) congruent descendants. By taking the topmost descendants, we ensure that all additional matching paths are unique, as at least one (different) e-class is chosen at each fork. Despite eliminating duplicate paths, some duplicate colored matches persist due to incomplete minimization of the e-graph. The modified instructions are described in more detail in Appendix B.

V. Evaluation

Support for colored e-graphs is implemented in a modified version of egg, called Easter Egg. In this section, we evaluate the performance and effectiveness of Easter Egg and the different optimizations we presented. For this purpose we implemented two versions of colored e-graphs containing different improvements described in section IV. The simple version only uses procedural improvements, while the optimized version uses all optimizations.

A. Objectives and Evaluation Method

Our evaluation aims to test colored e-graphs' efficacy in equality saturation for exploratory reasoning tasks with multiple simultaneous assumptions. We evaluate the effectiveness using e-graph size and equality saturation time. To the best of our knowledge, a purely e-graph-based automated theorem prover does not exist, and theory exploration tools have limited support for conditions. Thus, for the evaluation, we created an equality saturation-based prover (based on code from [26]) that incorporates an automatic case-splitting mechanism.

The case-splitting mechanism is only used when it will potentially contribute to progress of the equality saturation process—that is, when it enables additional rewrite rules that were previously blocked. When this is detected, the prover yields appropriate assumptions, one for each case. We compare two settings: a baseline setting with separate e-graphs created by cloning, and Easter Egg's colored e-graph implementation.

We measure the total running times and the total size of all the e-graphs.

We evaluated our implementation on inductive proof suites from [24], also used in [26]. Since the instances are relatively small, we introduced a slight variation: for each goal, we combined benchmarks (i.e. proof goals) within the suite sharing similar goals and vocabulary. This approach generates larger benchmarks, and thus larger e-graphs, for more significant exploration, with the prover continuing until saturation or resource limit, regardless of early goal achievement. All the experiments were conducted on 64 core AMD EPYC 7742 processor with 512 GB RAM.

B. Experimental Setup

Using the enhanced prover, we evaluated each test case by measuring e-graph sizes and run times. E-graph size was determined by counting e-nodes; in colored layers, we tracked additional colored e-nodes, whereas for separate e-graphs, we measured the e-nodes in both the original and coarsened graphs. The experiments utilize the Cap library to cap memory usage at 32 GB and limit run-time to 1 hour per case.

Our experiments involved a basic colored e-graph implementation (as per section II which we dub monochrome colored e-graph, as it does not contain colored e-nodes) and a fully optimized version, comparing both against the baseline of separate e-graphs. The pruning optimization has almost identical results to the fully optimized version, and hence, for brevity, it is not shown. It is expected, due to pruning being ineffective in cases where the same rewrite rules are applied repeatedly, adding the removed e-nodes right back.

C. Results

In our setup, all assumptions emerge from case splits done by the prover. We filter out cases where no case splits were applied, since these have no assumptions introduced and thus colored e-graphs have no impact.

For each benchmark instance, we measure the *relative e-node overhead* as the number of additional e-nodes that are required, normalized by the number of different assumptions. That is, $(|\text{total e-nodes}| - |\text{base e-nodes}|)/|\text{assumptions}|$. "Base e-nodes" represent the contents of the graph before case splits. (For the monochrome colored e-graph we use the base e-nodes present in the separate e-graphs case.) Figure 3 summarizes the results, pitting colored e-graphs (with and without colored e-nodes) against the baseline of separate clones. In some cases one configuration times out or runs out of memory, while the other does not; we only compare cases where both configurations finished the run successfully. In both comparisons, we see roughly around $10\times$ lower overhead, where in the monochromatic case samples are more dispersed around the y axis, and the optimized case shows clear advantage to the colored e-graph implementation.

Run-time is measured as the the total run-time for completed test cases, and 1 hour for cases that timed out. We do not include runs that did not finish due to out-of-memory exceptions (we report the latter separately). As can be seen in Figure 4, the monochrome colored e-graph lead to many timeouts, whereas the optimized case exhibits running times similar to separate clones. This is in line with our expectation: colors provide lower memory sizes at the expense of run-time.

TABLE I
RUN-TIME AND EXCEPTIONS. M = OUT OF MEMORY, T = TIMEOUT (3600)

	Separate		Monochrome		Optimized	
Test Suite	Time	M/T	Time	M/T	Time	M/T
clam	70.1	0/0	277.8	0/5	23.6	0/0
hipspec-rev-equiv	34.1	0/0	139.0	0/17	57.0	0/0
hipspec-rotate	3880.3	1/1	1871.4	0/6	17.4	0/3
isaplanner	8454.4	0/60	6068.4	0/70	20486.3	3/28
leon-amortize-queue	187356.4	52/0	14.8	0/57	10854.3	3/49
leon-heap	1735.9	0/0	1201.8	0/25	4949.2	0/13

Finally, in Table I we present the number of out-of-memory exceptions, the number of timeout exceptions, and total run-time for each configurations and test suite. The monochrome colored e-graph, as expected, exhibits many timeouts. Even though it has more errors than the other e-graph versions, it still has much longer run-times.

The optimized e-graphs demonstrate enhancements over separate e-graphs in both run-time and success rate, as detailed in Table I. Notably, the optimized configuration completed more tests (99 failures compared to 114). A key shift observed is the replacement of out-of-memory errors with timeouts, particularly in the leon-amortize-queue suite. However, leon-heap posed challenges for colored e-graphs, incurring 13 extra timeouts even in the optimized version. Conversely, the isaplanner suite showed a notable improvement, halving the failure rate in the optimized version compared to the baseline.

VI. Related Work

Theory exploration and its applications. Interest in exploratory reasoning in the context of functional calculi started with IsaCoSy [13], a system for lemma discovery based in part on CEGIS [28]. In a seminal paper, QuickSpec [27] propelled applicability of such reasoning for inferring specifications from implementations based on random testing, with deductive reasoning to verify generated conjectures [6], [12]. TheSy [26] and Ruler [20] have both incorporated e-graphs to some extent in the exploration process: they are used to speed up equivalence reduction of the space of generated terms, and, in [26], also the filtering and qualification phases using symbolic examples. The evaluation of the latter shows quite clearly that case splitting is a major obstacle to symbolic exploratory reasoning, due to the large number of different cases and derived assumptions.

In the area of conditional rewrite discovery, Speculate [4] naturally builds on the techniques from QuickSpec and depends on property-based testing techniques to generate inputs that satisfy some conditions. SWAPPER [25] is a relatively early example of exploring using SyGuS with a data-driven inductive-synthesis approach with emphasis on finding rules

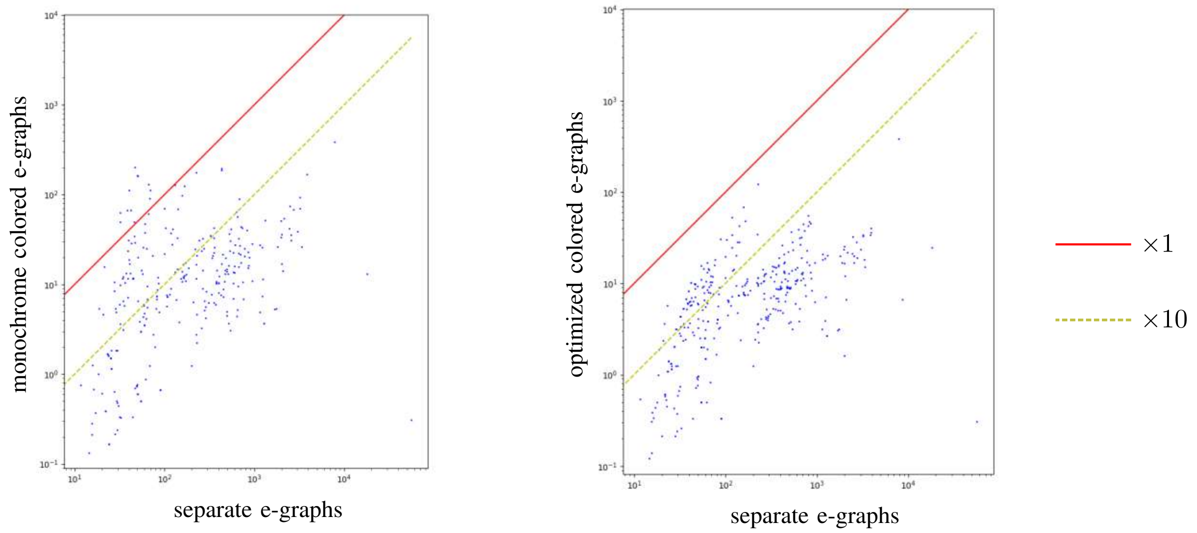


Fig. 3. Size comparison: relative e-node overhead in clones *vs.* color e-graph variants.

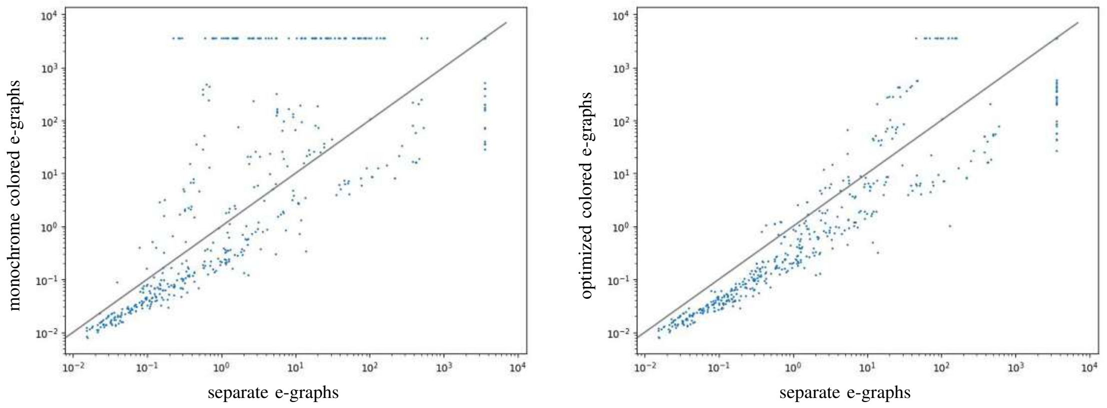


Fig. 4. Run-time comparison: run-time of clones vs. color e-graphs

that are most efficient for different problem domains. It requires a large corpus of similar SMT problems to operate.

Other e-graph extensions. E-graphs were originally brought into use for automated theorem proving [9], and were later popularized as a mechanism for implementing low-level compiler optimizations [29], by extending them with "φ-nodes" to express loops. Relational e-matching [32] makes use of Datalog seminaïve evaluation to harness the power of query planning in database systems. Subsequently, Datalog-powered e-matching has been recently fused with core Datalog semantics to allow richer logic programming by exposing equality saturation as a building block in a framework called egglog [31]. Since Datalog is based on Horn clauses, this meshes very well with conditional rewriting. It should be noted, though, that it is still a monotone framework, and does not allow backtracking or simultaneous exploration of alternative assumptions.

ECTAs [15], [11] are another, related compact data structure that extends e-graphs, Version-Space Algebras [17], [18], and Finite Tree Automata [1], with the concept of "entanglement"; that is, some choices of terms from e-classes may depend on choices done in other e-classes. Since the backbone of ECTAs is quite similar to an e-graph, the colors extension is applicable to this domain as well.

Uses of e-graphs in SMT. E-graphs are a core component for equality reasoning in SMT solvers [8], [2], in most theory solvers such as QF_UF, linear algebra, and bit-vectors. E-matching is also used for quantifier instantiation [21], which is, in its essence, an exploratory task and requires efficient methods [19]. In these contexts, implications and other Boolean structures are treated by the SAT core (in CDCL(T)), and the theory solver only handles conjunctions of literals.

VII. Conclusion

We presented colored e-graphs as an approach to efficiently handle multiple congruence relations in a single e-graph. They provide a memory-efficient method for equality saturation with additional assumptions, crucial for efficient exploratory reasoning of multiple assumptions simultaneously. Our optimizations, developed using the egg library, have shown notable improvements in memory usage and moderate enhancements in run-time performance over the baseline.

References

[1] Adams, M.D., Might, M.: Restricting grammars with tree automata. Proc. ACM Program. Lang. **1**(OOPSLA), 82:1–82:25 (2017). https://doi.org/10.1145/3133906, https://doi.org/10.1145/3133906

[2] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 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. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24

[3] Bergstra, J., Klop, J.: Conditional rewrite rules: Confluence and termination. Journal of Computer and System Sciences **32**(3), 323–362 (1986). https://doi.org/https://doi.org/10.1016/0022-0000(86)90033-4, https://www.sciencedirect.com/science/article/pii/0022000086900334

[4] Braquehais, R., Runciman, C.: Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In: Diatchki, I.S. (ed.) Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017. pp. 40–51. ACM (2017). https://doi.org/10.1145/3122955.3122961, https://doi.org/10.1145/3122955.3122961

[5] Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 350–367. Springer (2012). https://doi.org/10.1007/978-3-642-35182-2_25, https://doi.org/10.1007/978-3-642-35182-2_25

[6] Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: International Conference on Automated Deduction. pp. 392–406. Springer (2013)

[7] Coward, S., Constantinides, G.A., Drane, T.: Automating constraint-aware datapath optimization using e-graphs. In: 2023 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2023)

[8] De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

[9] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102, https://doi.org/10.1145/1066100.1066102

[10] Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchekha, P.: Small proofs from congruence closure. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 75–83. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13

[11] Gissurarson, M.P., Roque, D., Koppel, J.: Spectacular: Finding laws from 25 trillion programs. In: ICST. vol. 6. Association for Computing Machinery, New York, NY, USA (2023)

[12] Johansson, M.: Automated theory exploration for interactive theorem proving: - an introduction to the hipster system. In: Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. pp. 1–11 (2017). https://doi.org/10.1007/978-3-319-66107-0_1, https://doi.org/10.1007/978-3-319-66107-0_1

[13] Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. Journal of Automated Reasoning **47**, 251–289 (2010)

[14] Jones, E., Ong, C.H.L., Ramsay, S.: Cycleq: an efficient basis for cyclic equational reasoning. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 395–409 (2022)

[15] Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., Polikarpova, N.: Searching entangled program spaces. Proc. ACM Program. Lang. **6**(ICFP) (aug 2022). https://doi.org/10.1145/3547622, https://doi.org/10.1145/3547622

[16] Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: International Conference on Computer Aided Verification. pp. 1–35. Springer (2013)

[17] Lau, T.A., Domingos, P.M., Weld, D.S.: Version space algebra and its application to programming by demonstration. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000. pp. 527–534. Morgan Kaufmann (2000)

[18] Lau, T.A., Wolfman, S.A., Domingos, P.M., Weld, D.S.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1-2), 111–156 (2003). https://doi.org/10.1023/A:1025671410623, https://doi.org/10.1023/A:1025671410623

[19] de Moura, L.M., Bjørner, N.S.: 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, https://doi.org/10.1007/978-3-540-73595-3_13

[20] Nandi, C., Willsey, M., Zhu, A., Wang, Y.R., Saiki, B., Anderson, A., Schulz, A., Grossman, D., Tatlock, Z.: Rewrite rule inference using equality saturation. Proc. ACM Program. Lang. **5**(OOPSLA), 1–28 (2021). https://doi.org/10.1145/3485496, https://doi.org/10.1145/3485496

[21] 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, https://doi.org/10.1007/978-3-030-72013-1_8

[22] Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 65–74. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12

[23] Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Notices **50**(6), 1–11 (2015)

[24] Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 80–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[25] Singh, R., Solar-Lezama, A.: SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 185–192. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886678, https://doi.org/10.1109/FMCAD.2016.7886678

[26] Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 125–148. Springer (2021)

[27] Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. J. Funct. Program. **27**, e18 (2017). https://doi.org/10.1017/S0956796817000090, https://doi.org/10.1017/S0956796817000090

[28] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: 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. pp. 404–415 (2006). https://doi.org/10.1145/1168857.1168907

[29] Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 264–276. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480915, https://doi.org/10.1145/1480881.1480915

[30] Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434304, https://doi.org/10.1145/3434304

[31] Zhang, Y., Wang, Y.R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., Willsey, M.: Better together: Unifying datalog and equality saturation. In: PLDI '23: 44rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2023). https://doi.org/10.48550/arXiv.2304.04332, https://doi.org/10.48550/arXiv.2304.04332

[32] Zhang, Y., Wang, Y.R., Willsey, M., Tatlock, Z.: Relational e-matching. Proc. ACM Program. Lang. **6**(POPL), 1–22 (2022). https://doi.org/10.1145/3498696, https://doi.org/10.1145/3498696

APPENDIX A
BACKGROUND ON E-GRAPHS

We will now present some general background on e-graphs. Same as in section II, we assume a term language L where terms are constructed using *function symbols*, each with its designated arity. We use $f^{(r)} \in \Sigma[L]$ to say that f is in the *signature* of L and has arity r.

An e-graph $\mathcal{G}$ serves as a compact data structure representing a set $S \subseteq L$ of terms and a congruence relation $\cong \subseteq L \times L$. This congruence relation, in addition to being reflexive, symmetric, and transitive, is also closed under the function symbols of $\Sigma[L]$. That is, for every $f^r \in \Sigma[L]$, and given two lists of terms $t_{1..r} \in L$ and $s_{1..r}$, each of length r, if $t_i \cong s_i$ $(i = 1..r)$, then it follows that $f(t_1, \ldots, t_r) \cong f(s_1, \ldots, s_r)$. This property, known as *congruence closure*, is a key attribute of the data structure. The maintenance of this attribute as an invariant significantly influences the design and implementation of e-graph actions.

The egg library [30] revolutionizes the application of e-graphs by explicitly supporting the equality saturation workflow. It enables the periodic maintenance of congruence closure, via *deferred rebuild*, allowing for the amortization of associated rebuilding costs.

In egg, the authors present the e-graph as a union-find-like data structure, augmented to support operations on expressions. This implementation is primarily achieved through the utilization of three key structures: a hash-cons table, a union-find structure, and an e-class map. These structures collectively underpin the functionalities integral to the operation of the e-graph.

(a) The union-find component is responsible for keeping track of merged e-classes and maps each e-class id to a single representative for all (transitively) merged e-classes. This information is later used to canonicalize the keys and values of the hash-cons.
(b) The e-class map stores the structure of the e-graph. For each e-class id, the map keeps all the e-nodes that are contained therein. E-nodes are similar to AST nodes except that their children point to e-class ids instead of containing a single sub-term each.
(c) The hash-cons table maps e-nodes to their containing e-class id. An important aspect of the hash-cons is that after rebuilding, its keys and values are expected to be *canonical*. That is, whenever e-classes are merged one of their ids becomes "the" representative.

An e-class with id e represents a set of terms defined recursively as:

$$L(e) = \{f(t_1, .., t_k) \mid \\ f(e_1, .., e_k) \in M(e), t_i \in L(e_i) \text{ for } i = 1..k\}$$

We will use the notation $[t]$ to refer to e-class id where $t \in L([t])$.

Example A.1. The terms $\mathsf{max}(x, y)$ and $x - y$ are both represented in the e-graph in Figure 1(a) using e-classes $\langle 5\rangle$ and $\langle 6\rangle$, respectively, with the following e-nodes:

$$\begin{array}{rcll} M & = & \langle 1\rangle \mapsto \{\mathsf{true}\} & \langle 2\rangle \mapsto \{\mathsf{false}\} \\ & & \langle 3\rangle \mapsto \{x\} & \langle 4\rangle \mapsto \{y\} \\ & & \langle 5\rangle \mapsto \{\mathsf{max}(\langle 3\rangle, \langle 4\rangle)\} & \langle 6\rangle \mapsto \{\langle 3\rangle - \langle 4\rangle\} \end{array}$$

An e-graph where every e-class is a singleton, like this one, is just a forest of expression trees with sharing. The situation becomes more interesting once we start mutating the graph via its dedicated operations.

1) Insert - Adds a term t to the e-graph, one e-class per AST node, reusing e-classes where possible by searching the hash-cons.
2) Merge - Merging two e-classes by applying a union operation of the union-find and merging the classes in the e-class map. This, however, temporarily invalidates the invariant of the hash-cons and e-class map that all e-class ids and e-nodes must be canonical.
3) Rebuilding (Congruence closure) - As explained before, a union of $[x]$ into $[y]$ necessitates replacing any e-node $f([x], [z])$ by $f([y], [z])$. Moreover, if $f([x], [z]) \in [w_1], f([y], [z]) \in [w_2]$, then, following this replacement, both $[w_1]$ and $[w_2]$ now contain $f([y], [z])$, meaning that $[w_1] = [w_2]$ and evoking a cascading union of $[w_1], [w_2]$. A significant contribution by egg is the concept of deferred (and thus periodic) rebuilding. This periodic rebuilding is highly efficient and well-suited for equality saturation.
4) E-matching - Looking up a *pattern* in the set of terms represented by the e-graph in a top-down manner, traversing the e-nodes downward via the e-class map. A pattern is a term with (zero or more) *holes* represented by metavariables $?v_{1..k}$. For example, $(?v_1 + 1) \cdot ?v_2$ is a pattern. Pattern lookup is important for rewriting in equality saturation.

Rewriting. We assume a background set of symbolic *rewrite rules* (r.r.), each of the form $t \dot{\to} s$, where t and s are patterns as explained in item (4) above. A *match* θ of pattern t on the e-graph, is an assignment mapping metavariables to e-class ids. $t\theta$ represents an e-node, and we will denote its equality class as $[t\theta]$. Applying the r.r. is done by merging the e-classes $[s\theta]$ and $[t\theta]$. Because the e-node $s\theta$ might be new, it needs to also be inserted, resulting in $union([t\theta], insert(s\theta))$. Repetitively applying such rewrite rules to a set of terms can be used to generate growing sets of terms that are equivalent, according to rewrite semantics, to ones in the starting set. Ideally, the set eventually *saturates*, in which case the e-graph now describes *all* the terms that are rewrite-equivalent. We point out that in many situations, the e-graph keeps growing as a result of rewrites and never gets saturated—so the number of successive rewrite iterations, or "rewrite depth", has to be bounded.

A *conditional rewrite rule* (c.r.r.) [3] is a natural extension of a r.r. that has the following form: $\varphi \Rightarrow t \dot{\to} s$ where φ is a precondition for rewriting t to s. For example, the rules for max are: $?x > ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?x$ and $?x \leq ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?y$. The semantics of a precondition φ is defined such that a term matching the pattern of φ must be unified with Boolean true in order for the rewrite to be applied.

APPENDIX B
ALGORITHMS PSEUDO CODE

Colored e-graphs introduce a few algorithmic changes to the operations of a normal e-graph. Here we present pseudo code for the important changes presented in the paper. Algorithm 1 presents the changes being made to the e-matching abstract machine to support *unoptimized* colored e-matching as presented in section III.

Algorithm 1 Instructions: compare and colored_jump

```
1: function COMPARE($i, j$)
2:     if $find(color, reg[i]) \neq find(color, reg[j])$ then
3:         backtrack
4:     end if
5: end function
6:
7: function COLORED_JUMP($i$)
8:     $siblings \leftarrow \{e | e \in E \wedge e \equiv_{color} eclass\}$
9:     for $sibling$ in $siblings$ do
10:        $reg[i] = sibling$
11:        $bs$.push(current_state)
12:    end for
13:    backtrack
14: end function
```

The rebuilding algorithm is also updated to accommodate for colored e-graphs in section III, and the pseudo code in addition to some explanations is presented here. We update the auxiliary function REPAIR to work on colored e-classes, and introduce two new helper functions: COLLECT_PARENTS and UPDATE_HASHCONS, as presented in Algorithm 2. COLLECT_PARENTS extract the parents of a colored e-class by combining the sets of parents of all the (root) e-classes contained therein. UPDATE_HASHCONS is used to make sure that the hashcons entries are in canonical forms. It was already a part of REPAIR in egg; it is only repeated here to point out that it only updates the hashcons for the root color, since no canonization is required for colored layers.

The pseudo code for the optimized e-matching instructions that were presented in section IV are presented in Algorithm 4.

APPENDIX C
WALKTHROUGH FOR EXAMPLE 2

This is the full walkthrough of the example in Figure 1 from the overview.

We walk through the steps needed to carry out the case splitting shown in Figure 2. The system contains the conditional rewrite rules shown on the right of Figure 5, which constitute the definitions of max and min, plus some prior knowledge about $|\cdot|$ and $-$.

Algorithm 2 Colored Rebuilding

```
1: function REBUILD
2:     for $color$ in $self.colors$ do
3:         while $self.worklist(color).len() > 0$ do
                ▷ empty the worklist into a local variable
4:             $todo \leftarrow$ TAKE($self.worklist(color)$)
                ▷ canonicalize and deduplicate the eclass refs to save calls to repair
5:             $todo \leftarrow \{self.find(color, eclass) \mid eclass \in todo\}$
6:             for each $eclass$ in $todo$ do
7:                 SELF.REPAIR($color, eclass$)
8:             end for
9:         end while
10:    end for
11: end function
12:
13: function REPAIR($color, eclass$)
14:     $parents \leftarrow$ COLLECT_PARENTS($color, eclass$)
15:     UPDATE_HASHCONS($color, parents$)
            ▷ deduplicate the parents; note that equal parents get merged and put on the worklist
16:     $new\_parents \leftarrow \{\}$
17:     for each $(p\_node, p\_eclass)$ in $parents$ do
18:         $p\_node \leftarrow self.canonicalize(color, p\_node)$
19:         if $p\_node$ is in $new\_parents$ then
20:             $self.merge(color, p\_eclass, new\_parents[p\_node])$
21:             $new\_parents[p\_node] \leftarrow self.find(color, p\_eclass)$
22:         end if
23:     end for
24:     if $color = \varnothing$ then
25:         $eclass.parents \leftarrow new\_parents$
26:     end if
27: end function
```

The semantics of a conditional rewrite rule in the domain of an e-graph is that the condition pattern should be matched and its root must be in the same e-class as true, and, additionally, the left-hand side should be matched as normal. For simplicity of presentation, we pretend that $\neg$ is a special case were the negated condition is e-matched and the e-class should contain false.

Starting with the base graph, Figure 2(a), we describe the operation of Easter Egg on the red color, corresponding to the case $\neg x < y$. The complement blue case $(x < y)$ is analogous.

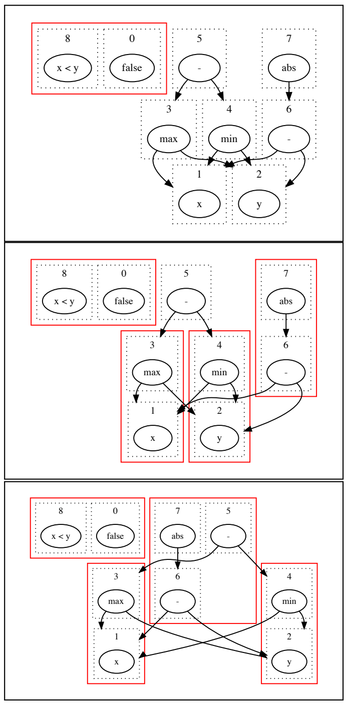
1) The value of $x < y$ is declared as false via a colored merge. This yields a new red e-class.
2) Colored e-matching is performed against the premise of the c.r.r. $\neg ?x < ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?x$. The condition of the rule, $?x < ?y$, matches against the class $[x < y]$, which is indeed in the same red e-class as false.
Similar e-matches are carried out for the rules $\neg ?x < ?y \Rightarrow \mathsf{min}(?x, ?y) \dot{\rightarrow} ?y$ and $\neg ?x < ?y \Rightarrow |?x - ?y| \dot{\rightarrow}$

(1)
$\text{merge}_{\color{red}\text{r}}([x < y],\ [\mathsf{false}])$

(2)
$\text{merge}_{\color{red}\text{r}}([\mathsf{max}(x, y)],\ [x])$
$\text{merge}_{\color{red}\text{r}}([\mathsf{min}(x, y)],\ [y])$
$\text{merge}_{\color{red}\text{r}}([|x - y|],$
$[x - y])$

(3)
$\text{rebuild}_{\color{red}\text{r}}()$
$\downarrow$
$\text{merge}_{\color{red}\text{r}}([x - y],$

$\mathsf{max}(x) - \mathsf{min}(y))$



rewrite rules

$?x < ?y \Rightarrow \mathsf{min}(?x, ?y) \dot{\rightarrow} ?x$
$\neg ?x < ?y \Rightarrow \mathsf{min}(?x, ?y) \dot{\rightarrow} ?y$
$?x < ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?y$
$\neg ?x < ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?x$
$?x < ?y \Rightarrow |?x - ?y| \dot{\rightarrow} ?y - ?x$
$\neg ?x < ?y \Rightarrow |?x - ?y| \dot{\rightarrow} ?x - ?y$

Fig. 5. Rewriting with case-split in a colored e-graph.

Algorithm 3 Colored Rebuilding (auxiliary methods)

```
function UPDATE_HASHCONS($color, parents$)
    if $color = \varnothing$ then
        for each $(p\_node, p\_eclass)$ in $parents$ do
            $self.hashcons.remove(p\_node)$
            $p\_node \leftarrow self.canonicalize(color, p\_node)$
            $self.hashcons[p\_node] \leftarrow self.find(color, p\_eclass)$
        end for
    end if
end function

function COLLECT_PARENTS($color, eclass$)
    $all\_parents \leftarrow \emptyset$ ▷ Initialize an empty set for parents
    $relevant\_eclasses \leftarrow \{e \mid e \in E \wedge e \equiv_{color} eclass\}$
    for $e$ in $relevant\_eclasses$ do
        $all\_parents \leftarrow all\_parents \cup e.parents$ ▷ Add parents of e to the set
    end for
    return $all\_parents$
end function
```

$?x - ?y$.

3) The children of $\langle 3\rangle - \langle 4\rangle$ ($\in M(\langle 5\rangle)$) are red-equivalent to those of $\langle 1\rangle - \langle 2\rangle$ ($\in M(\langle 6\rangle)$), and, as a consequence, red congruence closure kicks in and performs a red union there.

The process for blue is analogous. The case-split semantics is defined such that it records the fact that blue and red are *complements*, and as such extends $\equiv$ with the common equivalences, ${\color{blue}\cong_{\text{b}}} \cap {\color{red}\cong_{\text{r}}} = \{\langle\langle 5\rangle, \langle 7\rangle\rangle, \ldots\}$.

Algorithm 4 Instructions: optimized compare and colored_jump

```
1: function COMPARE'($i, j$)
2:     if $find(color, reg[i]) \neq find(color, reg[j])$ then
3:         $descendants \leftarrow \{c \mid color \in p^+(c) \wedge reg[i] \equiv_c reg[j]\}$
4:         $minimal \leftarrow \{c \mid c \in descendants \wedge \neg\exists c' \in descendants.\ c' \in p^+(c)\}$
5:         for $c$ in $minimal$ do
6:             $color = c$
7:             $bs$.push(current_state)
8:         end for
9:         backtrack
10:    end if
11: end function
12:
13: function COLORED_JUMP'($i$)
14:    $siblings \leftarrow \{e \mid e \in E \wedge e \equiv_{color} eclass\}$
15:    for $sibling$ in $siblings$ do
16:        $reg[i] = sibling$
17:        $bs$.push(current_state)
18:    end for
19:    $descendants \leftarrow \{(c, e) \mid color \in p^+(c) \wedge reg[i] \equiv_c e \wedge e \notin siblings\}$
20:    $minimal \leftarrow \{(c, e) \mid (c, e) \in descendants \wedge \neg\exists(c', e') \in descendants.(c' \in p^+(c) \wedge e' \equiv'_c e)\}$
21:    for $(c, e)$ in $minimal$ do
22:        $color = c$
23:        $reg[i] = e$
24:        $bs$.push(current_state)
25:    end for
26:    backtrack
27: end function
```

Formal Methods in Computer-Aided Design 2024

Word Equations as Abstract Domain for String Manipulating Programs

Antonina Nepeivoda
Program Systems Institute of RAS
Russia
a_nevod@mail.ru

***Abstract*—The paper presents a conceptual approach to abstract interpretation of string-manipulating programs, based on the existential theory of strings.**

We propose the word equation language as a base for lattices forming abstract domains of the string data. We construct a quantifier-free layer of the lattices, capturing the uniqueness properties of join and meet operations. The resulting finite-height lattice WL_0 utilizes useful properties of primitive roots of words and can be used as a base for future developments of word-equation-based abstract domains.

We describe a tokenization procedure as a monotone lattice mapping, in order to enhance expressiveness of word equation language by means of string morphisms and special cases of other finite-state-machine transformations.

***Index Terms*—program analysis, abstract interpretation, word equations, lattice mappings**

I. Introduction

The problem of static analysis of string manipulating programs, especially in dynamically typed languages, is known to be hard. For instance, even the theory with the replace-all and concatenation functions is undecidable [1], as well as the theory with the concatenation and letter counting operations [2].

Moreover, most of linear orders on the set of strings depend on the alphabet numeration. This fact makes construction of partition of the set of strings to polyhedra or intervals non-trivial and problem-specific ones.

In order to make the problem tractable, appropriate over-approximations and restrictions are used in static analysis [3]–[5]. In abstract interpretation, if a decidable set of predicates is taken as an abstract domain, the main two problems arise:

- how to over-approximate the wide variety of string operations in the string domain by the operations in the abstract domain;
- how to over-approximate the infinite chains of the predicates in the abstract domain by finite chains, in order to make the static analysis terminating.

The more precise are mappings into the abstract domain, the longer chains can occur; on the other hand, too small lattice height guarantees very fast convergence of the analysis, but may have drastically low preciseness of the analysis, compared with the methods admitting finite chains of non-uniformly bounded length.

The research was partially supported by Huawei Technologies Co. Ltd., and by Russian Academy of Sciences, research project №122012700089-0.

Thus, the main two approaches to construct the abstract domains exist. The first one considers some decidable fragment of string theory, and defines the appropriate generalizations (widening operations [6]) in order to collapse the infinite chains [6]–[9]. This approach is language-independent, allowing high flexibility of the tracked program properties, by varying the widening operation. The second one takes concrete practical properties of interest as the abstract domain, and solves concrete verification tasks in terms of the chosen programming language [10], [11]. The lattices used in this analysis are of small fixed height making the analysis fast.

For example, in ECMAScript language [12], the set of numbers is defined over a wider alphabet than $\{[0-9], ., -\}$. The constants `infinity` and `NaN` are also considered as numerical data. Thus, if the property "can represent numerical data" is tracked, then the approach making use of a string theory is forced to make the widening operator more precise, risking to make the whole analysis potentially slower.

In order to combine these two approaches, one can use an abstract domain in a decidable string theory, together with taking a quotient [13] wrt a partition of the string data set, taking into account language-specific properties of its elements. The partition can be defined as a preprocessing tokenization procedure, thus changing the underlying alphabet in the same string theory. Hence, the string manipulating operations are to be interpreted both by tokenization algorithm and the computations over the abstract domain. Under certain conditions, the tokenized strings and predicates on the tokens can happen to be fixed points of the lattice on the input string data, thus forming a proper finite sub-lattice [14]. Thus, a decidable string theory may be chosen in such a way that the tokenization procedure becomes in some sense "orthogonal" to predicates of the theory. It is known that the existential theory of words (the theory of word equations) is decidable [15], and the set of word equation languages neither contains nor is contained in the set of regular languages [16]. The set of word equation languages is not closed under morphic images and inverse morphic images [16]. Hence, the tokenization is able to significantly improve expressiveness of the given theory.

In order to address the widening problem, we advocate to use a natural string property known as the primitive root factorization, which is expressible in the word equation language. A root of a word ω is a ξ s.t. $\xi^n = \omega$. A word ξ is primitive iff $\forall \tau, n\big(\xi = \tau^n \Rightarrow n = 1\big)$. Word equations may encapsulate a

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_14

wide variety of properties including some of statements about primitive roots. E.g. the equation $XY = YX$ represents the predicate "if the strings X and Y are non-empty, then they have the same primitive root".

Let us show an example of how the notion of the primitive root helps to solve the widening problem. The set of the regular expressions, which is known to be closed under both intersection and union, forms a distributive lattice [6], [8]. However, the regular expressions admit infinite ascending chains, e.g. $\mathcal{L}(a) \subseteq \mathcal{L}(a|a^2) \subseteq \mathcal{L}(a|a^2|a^3) \subseteq \dots$, where $\mathcal{L}(r)$ denotes the language recognised by expression r. The most obvious widening is to define the widened value as the Kleene iteration a^*. Under this definition, the values a^* and $(a^2)^*$ are still distinct, and the latter implies the former. Thus, we can define an infinite descending chain of the predicates, i.e. a^*, $(a^2)^*$, …, $(a^{2^n})^*$, …, which violates the lattice finiteness condition. Now let us define the widened value Iter_a as a predicate satisfied by all words X satisfying the word equation $aX = Xa$. The definition for Iter_a using the word equations makes it possible to collapse the predicates to a single layer of the lattice. Since $\forall \tau, n(n > 0 \Rightarrow \tau a^n = a^n \tau \Leftrightarrow \tau a = a\tau)$, any predicate Iter_{a^n} is equivalent to the predicate Iter_a, thus the chains collapse to single elements.

The contributions of the paper are as follows.

First, we suggest a word equation language as a base for a lattice forming an abstract domain of string data. We construct a first (quantifier-free) layer of the lattice, capturing the uniqueness properties of joins and meets (Section III). The resulting finite-height lattice $\mathbf{WL_0}$ utilizes useful properties of primitive roots and can be used as a base for future developments of word-equation-based abstract domains.

Second, we suggest a tokenization procedure as a monotone lattice mapping, in order to enhance expressiveness of the word equation language by means of string morphisms and special cases of other finite-state-machine transformations, based on inverse mappings of the string morphisms (Section V). Moreover, given a string morphism onto the string set of the initial abstract domain, the set of its fixpoints forms a complete sublattice of the lattice $\mathbf{WL_0}$, and no additional construction is required to track additional program properties captured by the morphism.

The paper is organized as follows. In Section II, the main notions of lattice theory and word equation theory are given. In Section III, the experimental lattice based on the word equations is presented, and Section IV presents abstract domain semantics of the standard string operations used in ECMAScript programs. Section V considers the tokenization transformations, Section VI discusses related works, and Section VII concludes the paper.

II. Preliminaries

Small Greek letters (maybe with indices) stand for finite constant words (strings); domains and sets are denoted with Greek capitals. Small Latin letters a, b, c, d are considered to be elements of Σ. Capital Latin letters X, Y, Z stand for elements of the variable alphabet. The notation τ^n stands for n-concatenation of τ with itself, i.e. $\underbrace{\tau\tau\dots\tau}_{n}$. The empty word is denoted by ε. Given a word τ, $|\tau|$ stands for its length.

A word τ is said to be *primitive* (denoted with $\mathrm{prm}(\tau)$), if $\forall \xi, n\big(\tau = \xi^n \Rightarrow n = 1\big)$. Thus ε is not a primitive word, since $\forall n(\varepsilon^n = \varepsilon)$. Every non-empty word τ has a unique primitive root ξ, i.e. $\forall n, m, \xi, \xi'\big(\xi^n = \tau = (\xi')^m \;\&\; |\tau| > 0 \;\&\; \mathrm{prm}(\xi) \;\&\; \mathrm{prm}(\xi') \Rightarrow \xi = \xi'\big)$. We denote the primitive root of τ with $\rho(\tau)$.

Definition II.1. *Given a letter alphabet Σ and a variable alphabet Ξ,* a word equation *is an equation $\mathcal{U} = \mathcal{V}$, where $\mathcal{U}, \mathcal{V} \in (\Sigma \cup \Xi)^*$.*

A solution *to an equation $\mathcal{U} = \mathcal{V}$ is a morphism σ which is identity on Σ and maps elements of Ξ into Σ^*, s.t. $\sigma(\mathcal{U}) = \sigma(\mathcal{V})$ [15], [16].*

We also call the set of possible tuples of variable images determined by solutions of $\mathcal{U} = \mathcal{V}$ the solution set of $\mathcal{U} = \mathcal{V}$. Given a variable set $\mathcal{Q}$, the solution set of $\mathcal{U} = \mathcal{V}$ wrt $\mathcal{Q}$ is the projection of the solution set of $\mathcal{U} = \mathcal{V}$ on the coordinates corresponding to the elements of $\mathcal{Q}$.

The following examples are classical [17], [18].

Example II.1. *Given an equation $aX = Xa$, where $X \in \Xi$ and $a \in \Sigma$, its solution set is $\big\{a^n \mid n \in \mathbb{N}\big\}$.*

Given an equation $ZX = XY$, where $X, Y, Z \in \Xi$, its solution set for (X, Y, Z) is $\big\{((\xi_1\xi_2)^n\xi_1, \xi_2\xi_1, \xi_1\xi_2) \mid n \in \mathbb{N} \;\&\; \xi_1, \xi_2 \in \Sigma^*\big\}$. The solution set of $ZX = XY$ wrt the variable X is $\big\{(\xi_1\xi_2)^n\xi_1 \mid \xi_1, \xi_2 \in \Sigma^*\big\}$. It implies that, given an equation $\tau_1 X = X\tau_2$, its solution set is non-empty iff $\exists \eta_1, \eta_2\big(\tau_1 = \eta_1\eta_2 \;\&\; \tau_2 = \eta_2\eta_1\big)$.

Given an equation $\xi_1\xi_2X = X\xi_2\xi_1$, where $\xi_1, \xi_2 \in \Sigma^*$, $\mathrm{prm}(\xi_1\xi_2)$ and $|\xi_2| > 0$, and $X \in \Xi$, its solution set is $\big\{(\xi_1\xi_2)^n\xi_1 \mid n \in \mathbb{N}\big\}$. If $|\xi_1| = 0$, then the equation is reduced to $\xi_2X = X\xi_2$, and its solution set is $\big\{\xi_2^n \mid n \in \mathbb{N}\big\}$.

Let us denote the predicate "τ satisfies the equation $\xi_1\xi_2\tau = \tau\xi_2\xi_1$" with $\mathrm{Cnj}_{\xi_1,\xi_2}(\tau)$. We assume that the representation of $\mathrm{Cnj}_{\xi_1,\xi_2}$ is reduced by default to the shortest possible value of $\xi_1\xi_2$, i.e. the word $\xi_1\xi_2$ is primitive in $\mathrm{Cnj}_{\xi_1,\xi_2}$. Moreover, since $\mathrm{Cnj}_{\tau,\varepsilon} = \mathrm{Cnj}_{\varepsilon,\tau}$, we always choose $\mathrm{Cnj}_{\varepsilon,\tau}$ as a default.

We recall the following classical Fine–Wilf theorem [18].

Theorem II.1. *Let $\xi_1, \xi_2 \in \Sigma^+$. Suppose ξ_1^m and ξ_2^n, for some $m, n \in \mathbb{N}$, have a common prefix of length $|\xi_1| + |\xi_2| - gcd(|\xi_1|, |\xi_2|)$. Then there exists $\tau \in \Sigma^*$ of length $gcd(|\xi_1|, |\xi_2|)$ such that $\tau = \rho(\xi_1) = \rho(\xi_2)$, i.e. τ is the primitive root both of ξ_1 and ξ_2.*

Definition II.2. *A triple $\langle \mathcal{L}, \vee, \wedge \rangle$, where $\mathcal{L}$ is a set, $\vee$ and $\wedge$ are binary operations over $\mathcal{L}$ (also called join and meet respectively), is said to be a lattice if it satisfies the following axioms [19] for all $x, y, z \in \mathcal{L}$:*

- $\big(x \vee (x \wedge y) = x\big) \;\&\; \big(x \wedge (x \vee y) = x\big)$;
- $\big(x \vee y = y \vee x\big) \;\&\; \big(x \wedge y = y \wedge x\big)$;
- $\big(x \vee (y \vee z) = (x \vee y) \vee z\big) \;\&\; \big(x \wedge (y \wedge z) = (x \wedge y) \wedge z\big)$.

An order induced on a lattice E with the lattice operations is defined as follows: $x \le y \equiv (x \vee y = y)$.

Given lattices E, F, a mapping $\phi : E \to F$ is said to be consistent with the order (*isotonic*) iff $\forall x, y\big(x \le y \Rightarrow \phi(x) \le \phi(y)\big)$ [14]. A mapping ϕ is said to be *a lattice morphism* iff it respects both joins and meets [14].

The following lemma demonstrates a useful property of the equations $\xi_1\xi_2X = X\xi_2\xi_1$ (assuming by definition that $\mathrm{prm}(\xi_1\xi_2)$). Henceforth we call such equations elementary.

Lemma II.1. *If the words ξ_1, ξ_2, ξ_3, ξ_4 satisfy $\mathrm{prm}(\xi_1\xi_2)$ and $\mathrm{prm}(\xi_3\xi_4)$, and $\xi_1 \neq \xi_3$ or $\xi_2 \neq \xi_4$, then there exists at most one word $\tau \in \Sigma^*$ satisfying both $\mathrm{Cnj}_{\xi_1,\xi_2}(\tau)$ and $\mathrm{Cnj}_{\xi_3,\xi_4}(\tau)$.*

Proof. Let $\tau = (\xi_1\xi_2)^n\xi_1 = (\xi_3\xi_4)^m\xi_3$. Without loss of generality, we assume that $|\xi_1\xi_2| \ge |\xi_3\xi_4|$; the opposite case is symmetric.

If $n > 1$ (and hence $m > 1$), then the word $\tau\xi_2 = (\xi_1\xi_2)^{n+1}$ and the word $\tau\xi_4 = (\xi_3\xi_4)^{m+1}$ share a common prefix of the length $|\tau|$, which is at least $|\xi_1|+|\xi_2|+|\xi_3|+|\xi_4|$. Hence, by the Fine–Wilf theorem [20], $\xi_1\xi_2$ and $\xi_3\xi_4$ share a common primitive root, i.e. are equal, because they are primitive. Hence, $n = m$ and $\xi_1 = \xi_3$, which contradicts the choice of ξ_i.

Thus, if there are such distinct $\tau_0, \tau_1 \in \Sigma^*$, both belonging to the solution sets of $\xi_1\xi_2X = X\xi_2\xi_1$ and $\xi_3\xi_4X = X\xi_4\xi_3$, then $\tau_i = (\xi_1\xi_2)^i\xi_1$. I.e. $\exists k_1 \ge 0, k_2 > 0$ s.t. $\tau_0 = \xi_1 = (\xi_3\xi_4)^{k_1}\xi_3$, $\tau_1 = \xi_1\xi_2\xi_1 = (\xi_3\xi_4)^{k_1+k_2}\xi_3 = (\xi_3\xi_4)^{k_2}(\xi_3\xi_4)^{k_1}\xi_3 = (\xi_3\xi_4)^{k_2}\xi_1$. That implies $\xi_1\xi_2 = (\xi_3\xi_4)^{k_2}$, hence, $k_2 = 1$ and $\xi_1 = \xi_3$, since $\xi_1\xi_2$ is primitive, which contradicts the choice of ξ_i.

□

The proof above immediately implies the following Corollary. If Lemma's II.1 premise is true, then the only one of the following three cases can hold.

- No word satisfies the predicate $\mathrm{Cnj}_{\xi_1,\xi_2}$ & $\mathrm{Cnj}_{\xi_3,\xi_4}$.
- $\exists k\big(\xi_1 = (\xi_3\xi_4)^k\xi_3\big)$.
- $\exists k\big(\xi_1\xi_2\xi_1 = (\xi_3\xi_4)^k\xi_3\big)$.

We denote the word satisfying both $\mathrm{Cnj}_{\xi_1,\xi_2}$ and $\mathrm{Cnj}_{\xi_3,\xi_4}$ by $\mathrm{conjr}(\xi_1, \xi_2, \xi_3, \xi_4)$. Lemma II.1 shows that the predicates $\mathrm{Cnj}_{\xi_1,\xi_2}$ and $\mathrm{Cnj}_{\xi_3,\xi_4}$ are "orthogonal" wrt the sets of words satisfying them. For example, if $\xi_2 \neq \xi_4$ then $\mathrm{conjr}(\varepsilon, \xi_2, \varepsilon, \xi_4) = \varepsilon$.

III. The Lattice Construction

Let us introduce a relation $\propto$ between elements of the concrete string domain $\mathcal{S}^{\#}$ and an abstract domain Δ. A word τ satisfies a predicate P, where $\tau \in \mathcal{S}^{\#}$ and $P \in \Delta$, iff $\tau \propto P$.

The antimonotonous Galois connection defined by the relation $\propto$ determines the abstraction and concretisation operations wrt the abstract domain Δ.

As usual, the values $\top$ and $\bot$ represent the greatest and the least element of the lattice. The first level higher than $\bot$ (i.e. layer 1) captures the trivial word equations $X = \xi$, denoted by Eq_ξ, which is standard for the string abstract domains [9], [10]. As for the next lattice levels (layers whose numbers start with 2), we require them to satisfy the following properties.

- For any element P of a layer higher than the layer 1 (i.e., the layer of the trivial word equations), there is an infinite string set for which P holds (expressiveness).
- Given any two distinct elements P_1 and P_2 of the layer N, there is at most one predicate P of the layer $N+1$ s.t. $P_1 \Rightarrow P$ and $P_2 \Rightarrow P$ hold (unique join).
- Given any two distinct predicates P_1 and P_2 of the layer $N+1$, there is at most one predicate P of the layer N s.t. $P \Rightarrow P_1$ and $P \Rightarrow P_2$ hold (unique meet).

The first property guarantees that all the elements of the layer are expressible enough; the second property is required to define unique join elements, and the third is used to define unique meet elements. Obviously, the top element $\top$ satisfies all the three conditions.

A natural question arises: how can one introduce a partial word equation order that is able to distinguish the equations belonging to various levels? Given equations $\mathcal{U}_1 = \mathcal{V}_1$ and $\mathcal{U}_2 = \mathcal{V}_2$ in alphabet Σ s.t. $|\Sigma| \ge 1$, an equation whose solution set wrt the variables occurring in $\mathcal{U}_1$, $\mathcal{U}_2$, $\mathcal{V}_1$, $\mathcal{V}_2$ is a union of the solution sets of the given equations can be constructed with introducing 2 additional (fresh) variables (see [16], in the earlier work [21] a construction with 4 additional variables is given). On the other hand, an equation with the solution set representing the intersection of the two solution sets above can be constructed without any additional variable, provided that $|\Sigma| > 1$. Hence, the number of distinct variables in a given equation can be treated as a measure for its "generality", provided that the solution set of the equation is considered wrt a single variable X. With respect to this measure, the simplest equations depend only on X itself, i.e. are of the form P : $\xi_1X\xi_2X\ldots\xi_nX = X\xi'_1X\ldots X\xi'_m$, where $|\xi_1| > 0$. If $m \neq n$, then P has finitely many solutions; thus, the expressiveness requirement is satisfied[1] only if $m = n$. The lemma below shows that any equation P with infinitely many solutions is equivalent to an equation of the form $\mathrm{Cnj}_{\xi_1,\xi_2}$.

Lemma III.1. *The set of predicates of the form $\mathrm{Cnj}_{\xi_1,\xi_2}$, where the word $\xi_1\xi_2$ is primitive, satisfies all the three conditions given above. Any other quantifier-free predicate satisfying the conditions is equivalent to a predicate $\mathrm{Cnj}_{\xi_1,\xi_2}$ in the given set.*

Proof. Given an equation P of the form $\xi_1X\xi_2X\ldots\xi_nX = X\xi'_1X\ldots X\xi'_n$, let us assume that there exists its solution ω s.t. $|\omega| \ge |\xi_1|$. Then $\omega = \xi_1\omega'$, and ω' is a solution of the equation which arises from P in virtue of the substitution $X \mapsto \xi_1X'$. I.e., removing the ξ_1-prefixes on both sides of $P[X \mapsto \xi_1X']$, we obtain the equation P' : $\xi_1X'\xi_2\xi_1X'\ldots\xi_n\xi_1X' = X'\xi'_1\xi_1X'\ldots X'\xi'_n$ that is to be satisfied by ω'. As well as the initial equation P, equation P' has prefixes ξ_1X' and X' in its left- and right-hand sides, hence, the reasoning above can be repeated until $|\omega'| < |\xi_1|$.

[1]As shown in the paper [22], such equations have either at most 3 solutions or infinitely many solutions.

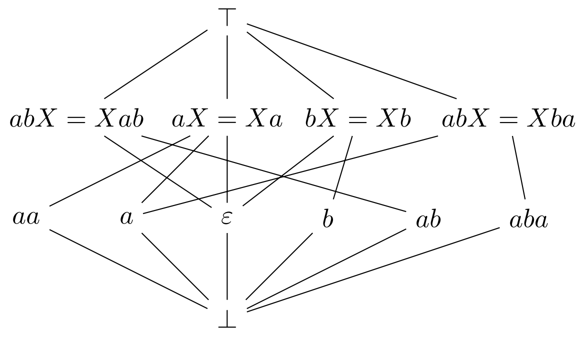


Fig. 1: Lattice built over constants ε, a, a^2, ab, aba. The values Eq_ξ are represented as ξ; the values $\mathrm{Cnj}_{\xi_1,\xi_2}$ are represented as the equations $\xi_1\xi_2X = X\xi_2\xi_1$.

Therefore, any solution to the equation P, where $|\xi_1| > 0$, is of the form $(\xi_{1,p}\xi_{1,s})^k\xi_{1,p}$, where $\xi_{1,p}\xi_{1,s} = \xi_1$.

Let us take such a number k_0 that $\max(\max_{1\le i\le n}|\xi_i|, \max_{1\le i\le n}|\xi_i'|) \cdot n < |\xi_1| \cdot k_0$, and separate the solution set of P into the following two sets.

- The words of the length less than $|\xi_1| \cdot k_0$.
- All the other words from the solution set of P. These words start with the prefix $\xi_1^{k_0}$; they can be seen as the solutions to equation $\sigma(P)$, where $\sigma : X \mapsto \xi_1^{k_0}X$.

Due to the choice of k_0, the equation $\xi_1 X\xi_2\xi_1^{k_0}X \dots \xi_n\xi_1^{k_0}X = X\xi_1'\xi_1^{k_0}X ... \xi_1^{k_0}X\xi_n'$ resulting from the mapping $X \mapsto \xi_1^{k_0}X$ can be split into n equations of the form $\tau_{i,1}X = X\tau_{i,2}$ (possibly, after reducing common prefixes and suffixes of the equation parts). Some of these equations are equivalent (if for some i, j and $k \in \{1, 2\}$ the primitive roots of $\tau_{i,k}$ and $\tau_{j,k}$ coincide), so we take only the subset of non-equivalent equations.

If this subset is a singleton, then the resulting equation is equal to the first equation $\xi_1X = X\tau$, where $|\tau| = |\xi_1|$, and τ may be either a prefix of ξ_1' (if $|\xi_1| < |\xi_1'|$) or of the form $\xi_1'\tau'$. In both cases, the solution set of this equation also includes any solution to P of the length less than $|\xi_1| \cdot k_0$.

If the set of the non-equivalent equations is not a singleton, then by Lemma II.1 the equation P has finitely many solutions and does not satisfy the expressiveness condition. Lemma II.1 guarantees that the unique meet condition holds.

Let us show that the unique join condition also holds. Given two distinct $\tau_1, \tau_2 \in \Sigma^*$ satisfying some elementary equation $\xi_1\xi_2X = X\xi_2\xi_1$ with ξ_1 and ξ_2 unknown, let $|\tau_1| > |\tau_2|$. Then $\exists\tau_3(\tau_3 \neq \varepsilon \ \&\ \tau_1 = \tau_3\tau_2)$, and the primitive root of τ_3 is equal to $\xi_1\xi_2$, while the suffix of τ_2 after the maximal prefix of the form $\rho(\tau_3)^k$ coincides with ξ_1. Hence, the values ξ_1 and ξ_2 in equation $\xi_1\xi_2X = X\xi_2\xi_1$ are determined by any two distinct words τ_1 and τ_2 satisfying this equation. $\square$

Lemma III.1 determines elements of the third level of lattice $\mathbf{WL_0}$, namely the set of predicates $\mathrm{Cnj}_{\xi_1,\xi_2}$ defining infinite solution sets of one-variable equations. A simple word-equation-based lattice can consist of the three given layers, and the top layer above them. Other possible extensions of the lattice are discussed in Section VII.

Based on the reasoning above, now we formally introduce the lattice elements and operations. The abstract domain Δ of the simplest lattice $\mathbf{WL_0}$ proposed in this paper consists of the following elements. As usual, we always assume that given a predicate $\mathrm{Cnj}_{\xi_1,\xi_2}$, the word $\xi_1\xi_2$ is primitive.

- Predicates Eq_ξ. $\mathrm{Eq}_\xi(\tau)$ iff $\tau = \xi$.
- Predicates "conjugates $\xi_1\xi_2$ and $\xi_2\xi_1$", denoted with $\mathrm{Cnj}_{\xi_1,\xi_2}$, where $|\xi_1\xi_2| > 0$. $\mathrm{Cnj}_{\xi_1,\xi_2}(\tau)$ iff $\xi_1\xi_2\tau = \tau\xi_2\xi_1$.
- The top element $\top$ representing all possible strings, and the bottom element $\perp$.

A simple example of such a lattice constructed over constants ε, a, a^2, ab, aba is presented in Fig. 1.

A. Operations of Lattice $\mathbf{WL_0}$

Let us define the join operation over the given domain. The right-hand sides of the definitions below are ordered to be applied from top to bottom.

- $\mathrm{Cnj}_{\tau_1,\tau_2} \vee \mathrm{Cnj}_{\xi_1,\xi_2} = \begin{cases} \mathrm{Cnj}_{\tau_1,\tau_2}, \text{ if } \forall i(\xi_i = \tau_i); \\ \top, \text{ otherwise;} \end{cases}$
- $\mathrm{Cnj}_{\tau_1,\tau_2} \vee \mathrm{Eq}_\xi = \begin{cases} \mathrm{Cnj}_{\tau_1,\tau_2}, \text{ if } \xi\tau_2\tau_1 = \tau_1\tau_2\xi; \\ \top, \text{ otherwise;} \end{cases}$
- $\mathrm{Eq}_\xi \vee \mathrm{Cnj}_{\tau_1,\tau_2} = \mathrm{Cnj}_{\tau_1,\tau_2} \vee \mathrm{Eq}_\xi$;
- $\mathrm{Eq}_{\tau_1} \vee \mathrm{Eq}_{\tau_2} = \begin{cases} \mathrm{Eq}_{\tau_1}, \text{ if } \tau_1 = \tau_2; \\ \mathrm{Cnj}_{\xi_1,\xi_2}, \text{ if } \exists \xi_1, \xi_2, k_1, k_2(k_1, k_2 \in \mathbb{N} \\ \ \&\ \tau_1 = (\xi_1\xi_2)^{k_1}\xi_1 \ \&\ \tau_2 = (\xi_1\xi_2)^{k_2}\xi_1); \\ \top, \text{ otherwise.} \end{cases}$.

The case returning $\mathrm{Cnj}_{\xi_1,\xi_2}$ as a value of $\mathrm{Eq}_{\tau_1} \vee \mathrm{Eq}_{\tau_2}$ reproduces the construction given in the proof of Lemma III.1, when the unique join property is checked.

E.g., $\mathrm{Eq}_\varepsilon \vee \mathrm{Eq}_a = \mathrm{Cnj}_{\varepsilon,a}$, as well as $\mathrm{Eq}_\varepsilon \vee \mathrm{Eq}_{aa} = \mathrm{Cnj}_{\varepsilon,a}$. $\mathrm{Eq}_{aba} \vee \mathrm{Cnj}_{a,b} = \mathrm{Cnj}_{a,b}$, since $(aba)ba = ab(aba)$.

The commutativity axiom for the join operation holds by definition.

If some of elements x, y, z of $\mathbf{WL_0}$ are equal, or any two of them are distinct equations, then the associativity $x \vee (y \vee z) = (x \vee y) \vee z$ also holds by definition. Let us consider the subtle case of the associativity: $x = \mathrm{Eq}_{\tau_1}$, $y = \mathrm{Eq}_{\tau_2}$, $z = \mathrm{Eq}_{\tau_3}$, $x \vee y = \mathrm{Cnj}_{\xi_1,\xi_2}$, $y \vee z = \mathrm{Cnj}_{\xi_3,\xi_4}$, $\xi_1 \neq \xi_3$ or

$\xi_2 \neq \xi_4$. Then by Lemma II.1 τ_2 is the only word satisfying the predicates $\mathrm{Cnj}_{\xi_1,\xi_2}$ and $\mathrm{Cnj}_{\xi_3,\xi_4}$ (i.e. $y \implies \mathrm{Cnj}_{\xi_1,\xi_2}$ & $y \implies \mathrm{Cnj}_{\xi_3,\xi_4}$), thus, $x \vee \mathrm{Cnj}_{\xi_3,\xi_4} = \top$ and $\mathrm{Cnj}_{\xi_1,\xi_2} \vee z = \top$ hold.

Now we define the meet operation.

- $\mathrm{Cnj}_{\tau_1,\tau_2} \wedge \mathrm{Cnj}_{\xi_1,\xi_2} = \begin{cases} \mathrm{Cnj}_{\tau_1,\tau_2}, \text{ if } \forall i(\xi_i = \tau_i); \\ \mathrm{Eq}_{\mathrm{conjr}(\tau_1,\tau_2,\xi_1,\xi_2)}, \\ \text{ if } \mathrm{conjr}(\tau_1,\tau_2,\xi_1,\xi_2) \text{ exists}; \\ \bot, \text{ otherwise.} \end{cases}$
- $\mathrm{Cnj}_{\tau_1,\tau_2} \wedge \mathrm{Eq}_{\xi} = \begin{cases} \mathrm{Eq}_{\xi}, \text{ if } \tau_1\tau_2\xi = \xi\tau_2\tau_1; \\ \bot, \text{ otherwise;} \end{cases}$
- $\mathrm{Eq}_{\tau} \wedge \mathrm{Cnj}_{\xi_1,\xi_2} = \mathrm{Cnj}_{\xi_1,\xi_2} \wedge \mathrm{Eq}_{\tau}$;
- $\mathrm{Eq}_{\tau_1} \wedge \mathrm{Eq}_{\tau_2} = \begin{cases} \mathrm{Eq}_{\tau_1}, \text{ if } \tau_1 = \tau_2; \\ \bot, \text{ otherwise.} \end{cases}$

There in the first case we refer to the property of elementary equations guaranteed by Lemma II.1. E.g., $\mathrm{Cnj}_{a,b} \wedge \mathrm{Cnj}_{\varepsilon,a} = \mathrm{Eq}_a$, since a satisfies both equations $abX = Xba$ and $aX = Xa$, hence, $a = \mathrm{conjr}(a, b, \varepsilon, a)$ (see Fig. 1).

By a similar reasoning, the $\wedge$ operation is associative.

Now we consider the last lattice condition to be checked.

- Since $\forall x, y\big(x \wedge y \Rightarrow x\big)$, and $\forall q\big((q \Rightarrow x) \Rightarrow (x \vee q = x)\big)$, the law $x \vee (x \wedge y) = x$ also holds.
- $x \wedge (x \vee y)$ is x iff $x \vee y \Rightarrow x$. The condition $x \vee y \Rightarrow x$ is guaranteed by the construction of the operations.

Hence the lattice definition is consistent. This lattice is not distributive. E.g. $\mathrm{Cnj}_{\varepsilon,a} \wedge (\mathrm{Cnj}_{\varepsilon,b} \vee \mathrm{Cnj}_{\varepsilon,c}) = \mathrm{Cnj}_{\varepsilon,a}$, but $(\mathrm{Cnj}_{\varepsilon,a} \wedge \mathrm{Cnj}_{\varepsilon,b}) \vee (\mathrm{Cnj}_{\varepsilon,a} \wedge \mathrm{Cnj}_{\varepsilon,c}) = \mathrm{Eq}_{\varepsilon}$.

IV. Operations on Lattice Elements

A. A Model Program

In order to demonstrate the computations in the abstract domain given above, let us consider the following example, given in a pseudocode (Fig. 2).

The $x + y$ concatenates x and y; $x - y$ deletes a prefix y from x; $prefix(x, y)$ checks whether a string x is a prefix of a string y.

```
1   z = $\xi$
2   x,y = $\varepsilon$
3   while ($cond_1$(i,j)) {        (depends only on $i$, $j$)
4     i = i + 1
5     x = x + z }
6   while ($cond_2$(i,j)) {        (depends only on $i$, $j$)
7     j = j + 1
8     y = y + z }
9   while (true) {
10    if ($prefix$(x,y))
11      y = y − x
12    elif ($prefix$(y,x))
13      x = x − y
14    else break }
```

Fig. 2: A fragment of a string-manipulating program incorrectly checking that a quotient of strings x and y is ε.

The program lines 9–14 aim at computing a “quotient” of the two strings, i.e. the word witnessing that the strings have different primitive roots. For example, if $x = abba$, $y = abbaab$, then the loop 9–14 breaks at the state $x = ba$, $y = ab$ after the two iterations. If the roots coincide, then the loop 9–14 is assumed to return ε, however if x is assigned to ε, the loop does not terminate. The reason of the non-termination is that ε is a prefix of any string, hence $\tau - \varepsilon = \tau$ for any $\tau \in \Sigma^*$. Moreover, the program given in Fig. 2 never terminates, because after executing lines 1–8 the values of x and y always have equal primitive roots.

Let us see how the corresponding operations are computed over the lattice $\mathbf{WL_0}$, and how the problem with the infinite loop can be revealed.

B. Computations in $\mathbf{WL_0}$

The following operations are chosen in order to demonstrate computations in $\mathbf{WL_0}$. The operations are analogous to operations included in standard string operating libraries, e.g. for ECMAScript [12]. Such a library includes at least concatenation operation, denoted with $x + y$; string replacement and truncation operations. In JavaScript, there exist the function replacing the first occurrence of a given string ξ in a string τ, and the function replacing all occurrences of ξ in τ. We denote the operation replacing the first occurrence of z_1 in y with z_2 with $replace(y, z_1, z_2)$. The string truncation usually depends on a given input — start and end positions of a substring that is to be deleted or extracted as an infix. We consider the following instance of the truncation: the string minus operation of the form $x - y$, where the prefix y is deleted from x.

We consider the versions of the operations with the numerical parameters unknown to the interpreter; if these parameters are known, the more precise over-approximations can be constructed. We assume that the right-hand sides of the interpretation rules in the interpretations given below are ordered from top to bottom to be applied. Some of the interpretations are straightforward; we comment only on the non-obvious ones. The order $\leq$ is induced by the join operation (see Section II). As usual, $\xi_1\xi_2$ in $\mathrm{Cnj}_{\xi_1,\xi_2}$ is assumed to be primitive.

Below the abstract version of the string concatenation is given.

$$x + y = \begin{cases} \mathrm{Eq}_{\xi_1\xi_2}, \text{ if } x = \mathrm{Eq}_{\xi_1}, y = \mathrm{Eq}_{\xi_2}; \\ \mathrm{Cnj}_{\xi_5,\xi_6}, \text{ if } x \leq \mathrm{Cnj}_{\xi_1,\xi_2} \text{ \& } y \leq \mathrm{Cnj}_{\xi_3,\xi_4} \\ \qquad \text{and } \xi_2\xi_1 = \xi_3\xi_4 \text{ and } \xi_5\xi_6 = \xi_1\xi_2 \\ \qquad\qquad \text{and } \exists n\big(\xi_1\xi_3 = (\xi_1\xi_2)^n\xi_5\big); \\ \top, \text{ otherwise.} \end{cases}$$

Given words τ_1 and τ_2 in the concrete string domain, if $\tau_1 + \tau_2 = (\xi_5\xi_6)^k\xi_5$, where k is large enough, then either $\tau_1 = (\xi_5\xi_6)^{k_1}\tau_5$ and $\tau_2 = \tau_6(\xi_5\xi_6)^{k_2}\xi_5$, where $\tau_5\tau_6 = \xi_5\xi_6$, or $\tau_1 = (\xi_5\xi_6)^k\tau_5$ and $\tau_2 = \tau_6$, where $\tau_5\tau_6 = \xi_5$. The case returning $\mathrm{Cnj}_{\xi_5,\xi_6}$ above includes both these instances. The parameter n above equals 0 if $|\xi_3| < |\xi_2|$, and equals 1 otherwise.

Below the abstract version of the string subtraction is given.

$$x - y = \begin{cases} \mathrm{Eq}_{\tau},\ \text{if } x = \mathrm{Eq}_{\xi\tau} \ \&\ y = \mathrm{Eq}_{\xi}; \\ \text{error, if } x = \mathrm{Eq}_{\tau} \ \&\ y = \mathrm{Eq}_{\xi} \ \&\ \forall\tau'(\tau \neq \xi\tau'); \\ \text{error, if } x = \mathrm{Cnj}_{\tau_1,\tau_2} \ \&\ y = \mathrm{Eq}_{\xi} \\ \qquad \&\ \neg(y \leq \mathrm{Cnj}_{\xi_1,\xi_2}),\ \text{where } \tau_1\tau_2 = \xi_1\xi_2; \\ x,\ \text{if } y = \mathrm{Cnj}_{\varepsilon,\xi} \text{ and} \\ \qquad \text{either} \quad x = \mathrm{Eq}_{\tau} \quad \text{and} \quad \forall\tau'\big(\tau \neq \xi\tau'\big), \\ \qquad \text{or } x = \mathrm{Cnj}_{\tau_1,\tau_2} \text{ and } \forall\tau', k\big((\tau_1\tau_2)^k\tau_1 \neq \xi\tau'\big); \\ \top,\ \text{if } y = \mathrm{Cnj}_{\xi_1,\xi_2} \text{ and } x - y \text{ can satisfy at least} \\ \qquad \text{two different predicates of the form} \,\mathrm{Cnj}_{\omega_1,\omega_2}; \\ \mathrm{Cnj}_{\tau_{1,2},\tau_2\tau_{1,1}},\ \text{if } x \leq \mathrm{Cnj}_{\tau_1,\tau_2} \text{ and } y \leq \mathrm{Cnj}_{\xi_1,\xi_2} \\ \qquad \text{and } \exists k, k', \tau_{1,1}, \tau_{1,2} \\ \qquad \big((\tau_1\tau_2)^k\tau_{1,1} = (\xi_1\xi_2)^{k'}\xi_1 \ \&\ \tau_1 = \tau_{1,1}\tau_{1,2}\big); \\ \mathrm{Cnj}_{\tau_{2,2}\tau_1,\tau_{2,1}},\ \text{if } x \leq \mathrm{Cnj}_{\tau_1,\tau_2} \text{ and } y \leq \mathrm{Cnj}_{\xi_1,\xi_2} \\ \qquad \text{and } \exists k, k', \tau_{2,1}, \tau_{2,2} \\ \qquad \big((\tau_1\tau_2)^k\tau_1\tau_{2,1} = (\xi_1\xi_2)^{k'}\xi_1 \ \&\ \tau_2 = \tau_{2,1}\tau_{2,2}\big); \\ \top,\ \text{otherwise.} \end{cases}$$

The x case above corresponds to the case when the prefix subtraction from x should succeed on the only possible concrete string value satisfying the predicate y. The intermediate case returning $\top$ corresponds to the case when a predicate of the form $\mathrm{Cnj}_{\tau_1,\tau_2}$ capturing concrete values of $x - y$ is undetermined, given arbitrary values satisfying the predicates – abstract values x and y. The remaining cases (besides the trivial one) consider the computations when such a predicate is unique. The detailed comments on the case with the undetermined value of $x - y$ and the cases returning conjugation predicates are given in Appendix (see Subsect. VII).

Now we consider the abstract version of the string replacement operation.

$$replace(y, z_1, z_2) = \begin{cases} \text{error, if } z_1 = \mathrm{Eq}_{\varepsilon}; \\ \mathrm{Eq}_{replace(\tau,\xi_1,\xi_2)}, \\ \ \text{if } y = \mathrm{Eq}_{\tau}, z_1 = \mathrm{Eq}_{\xi_1}, z_2 = \mathrm{Eq}_{\xi_2}; \\ \mathrm{Cnj}_{\tau_1,\tau_2},\ \text{if } y \leq \mathrm{Cnj}_{\tau_1,\tau_2} \\ \qquad \text{and } z_1, z_2 \leq \mathrm{Cnj}_{\varepsilon,\xi_1\xi_2} \text{ s.t. } \xi_2\xi_1 = \tau_1\tau_2; \\ \mathrm{Cnj}_{\tau_1,\tau_2},\ \text{if } y = \mathrm{Cnj}_{\tau_1,\tau_2} \\ \qquad \text{and } z_1 \leq \mathrm{Cnj}_{\xi_1,\xi_2} \text{ and } \forall n, k, \tau_3, \tau_4 \\ \qquad \Bigg(\big(\xi_1 \neq \varepsilon \Rightarrow (\tau_1\tau_2)^n \neq \tau_3\xi_1\tau_4\big) \\ \&\ \big(k > 0 \Rightarrow (\tau_1\tau_2)^n \neq \tau_3(\xi_1\xi_2)^k\xi_1\tau_4\big)\Bigg); \\ \top,\ \text{otherwise.} \end{cases}$$

There the satisfiability of y to the predicate $\mathrm{Cnj}_{\tau_1,\tau_2}$ is preserved in the following two cases. First, the result of the replacement satisfies $\mathrm{Cnj}_{\tau_1,\tau_2}$ if a power of a primitive word $\xi_1\xi_2$ conjugating with $\tau_1\tau_2$ (i.e. s.t. $\xi_2\xi_1 = \tau_1\tau_2$) is replaced with $(\xi_1\xi_2)^k$. Second, the value of y is unchanged if no occurrence of a string satisfying z_1 can appear in a string satisfying the predicate given by y.

C. Predicates

The predicates defined on the string domain, under certain conditions, may be equivalent to the predicates defined on some other domain, e.g. integers. For example, if $\exists z\big(y = xz\big)$, then we may deduce that $|x| \leq |y|$. If additionally x and y are known to belong to the language a^*, then the predicate $|x| \leq |y|$ becomes equivalent to $\exists z\big(y = xz\big)$. Thus, if the latter is replaced by the former, sometimes a dead code can be eliminated by a simple static analyser. On the other hand, if $\exists z\big(y = xz\big)$ and y is known to be ε, then the predicate $\exists z\big(y = xz\big)$ can be replaced by the equivalent condition $\big(y = \varepsilon\big)$ & $\big(x = \varepsilon\big)$, which can also simplify tracking some of unreachable computation branches.

In fact, the predicate processing searches for invariants of the conditionals or loops, that can be derived from the values of the variables involved in the predicates over the abstract domain. This technique is close to one used in the paper [23], in order to prune unreachable computation branches in string manipulating programs.

The following simple interpretation of the predicate $prefix(x, y) \Leftrightarrow \exists z(y = xz)$ helps a static analysis tool to detect the non-terminating loop shown in the program in Fig. 2. We denote a value of the concretisation function on the abstract value x with $a(x)$. There the line $prefix(x, y) = f\big(a(x), a(y)\big)$ is interpreted as "if intersection of the x-concretisation set and $Pref(y)$ is non-empty, where $Pref(y)$ is the set of prefixes of all elements of the y-concretisation set, then the predicate $prefix(x, y)$ can be replaced with $f\big(a(x), a(y)\big)$". The capitalized OR notation stands for the logical operation in the target program language.

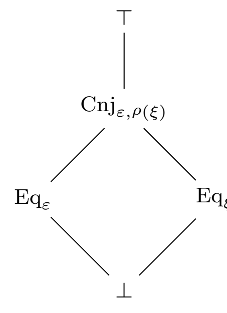
$$prefix(x, y) = \begin{cases} \big(a(x) = \xi_1 \text{ OR } \ldots \text{ OR } a(x) = (\xi_1\xi_2)^n\xi_1\big), \\ \qquad \text{if } x = \mathrm{Cnj}_{\xi_1,\xi_2} \\ \qquad \text{and } \exists\xi_3, n\big(n \in \mathbb{N} \ \&\ y = \mathrm{Eq}_{(\xi_1\xi_2)^n\xi_1\xi_3}\big); \\ |a(x)| \leq |a(y)|,\ \text{if } x = \mathrm{Cnj}_{\xi_1,\xi_2} \\ \qquad \text{and } y = \mathrm{Cnj}_{\xi_3,\xi_4} \text{ and } \xi_1\xi_2 = \xi_3\xi_4; \\ a(x) = \varepsilon,\ \text{if } y = \mathrm{Eq}_{\varepsilon}; \\ true,\ \text{if } \exists\xi\big(y = \mathrm{Eq}_{\xi_1\xi} \ \&\ x = \mathrm{Eq}_{\xi_1}\big); \\ false,\ \text{if } x = \mathrm{Eq}_{\xi_1}, y = \mathrm{Eq}_{\xi_2}\text{ , otherwise;} \\ \big(a(x) = \xi_1 \text{ OR } \ldots \text{ OR } a(x) = (\xi_1\xi_2)^n\xi_1\big), \\ \qquad \text{if } x = \mathrm{Cnj}_{\xi_1,\xi_2}, y = \mathrm{Cnj}_{\xi_3,\xi_4}, \exists\tau, m, n \\ \qquad \big(m, n \in \mathbb{N} \ \&\ (\xi_3\xi_4)^m\xi_3 = (\xi_1\xi_2)^n\xi_1\tau\big); \\ prefix\big(a(x), a(y)\big),\ \text{otherwise.} \end{cases}$$

The 1-st and the 6-th cases of the definition above contain a disjunction of n possible equalities for $a(x)$, which can be derived from the corresponding abstract values of x and y. The value of n is also determined by these abstract values. In the 6-th case n is bounded because $\xi_1\xi_2 \neq \xi_3\xi_4$ holds, since the case $\xi_1\xi_2 = \xi_3\xi_4$ is completely handled by the previous cases. In the 1-st case n is trivially bounded. Hence, the n-disjunction can be constructed without a loop.

A trace of the abstract interpretation using the interpretations given above is presented in Fig. 3. The notation $x \mapsto w$ states that the abstract value of x is w; $x \mapsto^* w$ states that the abstract value of x converges to w. In lines 5, 8, 11, 13, the fixed points of the computations are constructed. The join of $\mathrm{Eq}_{\varepsilon}$ and Eq_{ξ}, which is a value both of x and y, is $\mathrm{Cnj}_{\varepsilon,\rho(\xi)}$, and then $\mathrm{Cnj}_{\varepsilon,\rho(\xi)}$ is concatenated with Eq_{ξ} using the 2-nd rule for concatenation (Subsect. IV-B). The results of string subtraction stabilize in the same way. When the abstract interpretation converges, the predicates can be replaced in the concrete domain. After the replacement, a simple static

```
1   z = $\xi$                        $z \mapsto \mathrm{Eq}_\xi$
2   x,y = $\varepsilon$                    $x \mapsto \mathrm{Eq}_\varepsilon$, $y \mapsto \mathrm{Eq}_\varepsilon$
3   while (cond1(i,j)) {             (cond1(i,j) is outside the string domain)
4     i = i + 1                      (ignored)
5     x = x + z }                    $\mathrm{Eq}_\varepsilon \vee \mathrm{Eq}_\xi = \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$; $\mathrm{Cnj}_{\varepsilon,\rho(\xi)} \vee \mathrm{Cnj}_{\varepsilon,\rho(\xi)} = \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$, $x \mapsto^* \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$
6   while (cond2(i,j)) {             (cond2(i,j) is outside the string domain)
7     j = j + 1                      (ignored)
8     y = y + z }                    $\mathrm{Eq}_\varepsilon \vee \mathrm{Eq}_\xi = \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$; $\mathrm{Cnj}_{\varepsilon,\rho(\xi)} \vee \mathrm{Cnj}_{\varepsilon,\rho(\xi)} = \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$, $y \mapsto^* \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$
9   while (true) {
10    if (prefix(x,y))               (equivalent to $|x| \leq |y|$ after the interpretation)
11      y = y − x                    $\mathrm{Cnj}_{\varepsilon,\rho(\xi)} \vee \mathrm{Cnj}_{\varepsilon,\rho(\xi)} = \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$; $y \mapsto^* \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$
12    elif (prefix(y,x))             (equivalent to $|y| \leq |x|$ after the interpretation)
13      x = x − y                    $\mathrm{Cnj}_{\varepsilon,\rho(\xi)} \vee \mathrm{Cnj}_{\varepsilon,\rho(\xi)} = \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$; $x \mapsto^* \mathrm{Cnj}_{\varepsilon,\rho(\xi)}$
14    else break }                   (unreachable after the interpretation)
```

(a) Tracking the abstract values of the program variables.



$$\begin{tikzcd}
 & \top \arrow[d, no head] & \\
 & \mathrm{Cnj}_{\varepsilon,\rho(\xi)} \arrow[dl, no head] \arrow[dr, no head] & \\
\mathrm{Eq}_\varepsilon \arrow[dr, no head] & & \mathrm{Eq}_\xi \arrow[dl, no head] \\
 & \bot &
\end{tikzcd}$$

(b) The lattice on the abstract values used in the interpretation. Eq_ξ corresponds to the equation $X = \xi$; $\mathrm{Cnj}_{\xi_1,\xi_2}$ corresponds to the equation $\xi_1\xi_2X = X\xi_2\xi_1$.

Fig. 3: Static analysis of the program that incorrectly checks that a quotient of strings x and y is ε.

analysis tool can determine that the line 14 is unreachable because the disjunction $|x| \leq |y|$ OR $|y| \leq |x|$ always holds, and the loop given in the lines 9–14 never terminates.

V. Tokenization

In order to construct a sound mapping from the string set into a set of token sequences, in general we have to describe $\mathbf{WL_0}$-induced extensions of any finite state machine function. We postpone this problem to a future work, and now suggest a simple subclass of the finite-state-machine functions whose extensions are monotone lattice mappings.

Definition V.1. *Given alphabets Σ and Σ', let h be a string morphism being defined by the mapping $h' : \Sigma \to \Sigma'^*$. We use the same name h for the following extension of h over the lattice elements.*

- $h\big(\mathrm{Eq}_\xi\big) = \mathrm{Eq}_{h(\xi)}$
- $h\big(\mathrm{Cnj}_{\tau_1,\tau_2}\big) = \mathrm{Cnj}_{\rho\left(h(\tau_1)\right),\, \rho\left(h(\tau_1\tau_2) - \rho(h(\tau_1))\right)}$

We recall the following classical lemma [20], which ensures that h is monotonic wrt the lattice order.

Lemma V.1. *If σ is a solution to equation $\mathcal{U} = \mathcal{V}$, and h is a morphism, then $h \circ \sigma$ is a solution to $h\big(\mathcal{U}\big) = h\big(\mathcal{V}\big)$.*

Given any values $x, y \in \mathbf{WL_0}$ and a string morphism h, we can now show that $(h(x) \vee h(y)) \leq h(x \vee y)$. Moreover, in case $x \vee y \neq \top$, $h(x \vee y) = h(x) \vee h(y)$, due to Lemma V.1.

Let $\preceq$ be a linear order on Σ, and $\preceq_*$ be the length-lexicographical[2] order on Σ^* induced by $\preceq$. Given a string morphism $h : \Sigma \to \Sigma'^*$ s.t. $\forall a \in \Sigma\big(h(a) \neq \varepsilon\big)$, we define its minimal inverse mapping $h^{-1}_{\min} : \Sigma'^* \to \Sigma^*$ as follows.

$$h^{-1}_{\min}(\xi) = \tau \text{ s.t. } h(\tau) = \xi \;\&\; \forall \tau'\big(h(\tau') = \xi \Rightarrow \tau \preceq_* \tau'\big)$$

In general, the inverse image $h^{-1}_{\min}$ does not respect the lattice order. For example, given distinct $a \succ b \succ c$, if $h(c) = aba$, $h(a) = a$, $h(b) = b$, then $h^{-1}_{\min}(abab) = cb$, and $h^{-1}_{\min}\big(\mathrm{Eq}_{abab}\big) \vee h^{-1}_{\min}\big(\mathrm{Eq}_{ab}\big) = \top$, while $h^{-1}_{\min}\big(\mathrm{Eq}_{abab} \vee \mathrm{Eq}_{ab}\big) = h^{-1}_{\min}\big(\mathrm{Cnj}_{\varepsilon,ab}\big)$.

In order to address the monotonicity issue, we choose a special subset of string morphisms whose inverse mappings extensions can be used as lattice morphisms. After the paper [16], we say that an infix ξ of a word $a_1 \dots a_n$ contains a border between subwords $a_1 \dots a_k$ and $a_{k+1} \dots a_n$, if the word ξ includes an infix $a_{k-j_1} \dots a_{k+j_2}$, where $j_1 \geq 0$ and $j_2 > 0$. Given any predicate $\mathrm{Cnj}_{\xi_1,\xi_2}$ in lattice $\mathcal{L}$, the inverse mappings we consider preserve the borders between ξ_1 and ξ_2, as well as between ξ_2 and ξ_1.

Formally, given a lattice $\mathcal{L}$, the inverse mapping of a morphism h is border-preserving wrt $\mathcal{L}$, if for any lattice element of the form $\mathrm{Cnj}_{a_1 \dots a_k, a_{k+1} \dots a_n}$ and for any $b \in \Sigma'$, the morphism $h(b)$ is equal neither to $\xi_2(a_1 \dots a_n)^m \xi_1$ nor to ξ_3, for any $m \in \mathbb{N}$, $\xi_1, \xi_2, \xi_3 \in \Sigma^*$ satisfying the following conditions:

- ξ_1 is a prefix of $a_1 \dots a_n$; ξ_2 is a suffix of $a_1 \dots a_n$; $|\xi_1\xi_2| > 0$, $|\xi_i| < n$, and either $(|\xi_1| > 0)$ & $(|\xi_2| > 0)$, or $m > 0$,
- ξ_3 is $a_{k-j_1} \dots a_{k+j_2}$, where $k > j_1 \geq 0$ and $n - k \geq j_2 > 0$, and $|\xi_3| < n$.

Hence, there are the two possibilities to violate the border-preserving condition: $h(b)$ equals to an infix of $a_1 \dots a_n$ containing the border between $a_1 \dots a_k$ and $a_{k+1} \dots a_n$, or to a subword of $(a_1 \dots a_n)^m$ containing at least one border between the occurrences of $a_1 \dots a_n$. The border-preserving condition does not depend on the order induced on Σ in the definition of $h^{-1}_{\min}$, because the condition holds for images of all elements of Σ'.

Example V.1. *Given a lattice including the element $\mathrm{Cnj}_{\varepsilon,ab}$ and $c \neq a$, $c \neq b$, the inverse of any morphism h s.t. $\exists c \in \Sigma'\big(h(c) = aba\big)$ is not border-preserving: $h(c)$ includes the border between two occurrences of ab.*

[2]If $|\omega_1| < |\omega_2|$, then $\omega_1 \preceq_* \omega_2$; if $|\omega_1| = |\omega_2|$, then the order $\preceq$ is used lexicographically.

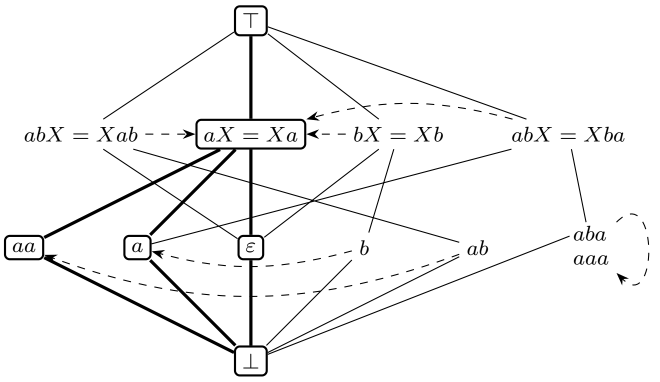


Fig. 4: Isotonic lattice mapping using the string morphism $h_{\Sigma \to a}$ of the lattice given in Fig. 1. The sublattice of the fixed points of $h_{\Sigma \to a}$ is given in framed nodes and thick edges. Dashed arcs point to the morphic images of the elements.

Given a lattice including the element $\mathrm{Cnj}_{aba,ba}$*, neither of the inverses of morphisms* h_1*,* h_2 *s.t.* $\exists c_1, c_2 \in \Sigma' (h_1(c_1) = bab \ \&\ h_2(c_2) = aa)$ *is border-preserving. The* bab *value of* $h_1(c_1)$ *includes the border between* aba *and* ba*, given* $\xi_3 = bab$*,* $k = 3$*,* $j_1 = 1$*,* $j_2 = 1$*; the* aa *value of* $h_2(c_2)$ *contains the border between* ba *and* aba*.*

Lemma V.2. *Given lattice* $\mathbf{WL_0}$ *and a string morphism* h *satisfying the conditions above, for any order induced on* Σ*,* $h^{-1}_{\min}$ *is a lattice morphism.*

Proof. Given $x, y \in \mathcal{L}$, let us show that $h^{-1}_{\min}(x \vee y)$ and $h^{-1}_{\min}(x) \vee h^{-1}_{\min}(y)$ are equal.

If x and y are both of the form $\mathrm{Cnj}_{\tau_i,\tau_j}$, then their join is non-trivial iff $x = y$; and the equality is trivially preserved.

Given $x = \mathrm{Eq}_{\xi_1}$ and $y = \mathrm{Eq}_{\xi_2}$, if they are equal or their join is $\top$, the morphism property holds trivially. Let us consider the case when $x \vee y$ is $\mathrm{Cnj}_{\tau_1,\tau_2}$, then for any $b \in \Sigma', \xi'$ s.t. $h(b) = \xi'$ and $\xi_i = \xi_{i,p}\xi'\xi_{i,s} = (\tau_1\tau_2)^{k_i}\tau_1$, the occurrence of ξ' can appear strictly inside the subwords τ_1 and τ_2 of ξ_1 and ξ_2. Thus $h^{-1}_{\min}\big(\mathrm{Eq}_{\xi_1} \vee \mathrm{Eq}_{\xi_2}\big) = \mathrm{Cnj}_{h^{-1}_{\min}(\tau_1), h^{-1}_{\min}(\tau_2)} = h^{-1}_{\min}\big(\mathrm{Eq}_{\xi_1}\big) \vee h^{-1}_{\min}\big(\mathrm{Eq}_{\xi_2}\big)$.

Given $x = \mathrm{Eq}_{\xi}$ and $y = \mathrm{Cnj}_{\tau_1,\tau_2}$, let us compute $h^{-1}_{\min}\big(x \vee y\big)$ when the join is non-trivial. In this case $\xi = (\tau_1\tau_2)^k\tau_1$, and again all the subwords ξ' s.t. $\exists b \in \Sigma'(h(b) = \xi')$ can occur only inside τ_1 or τ_2, thus the resulting join is $\mathrm{Cnj}_{h^{-1}_{\min}(\tau_1), h^{-1}_{\min}(\tau_2)}$.

The meet case is symmetric to the join case. Note that both by the definition of $h^{-1}_{\min}$ and choice of h, $h^{-1}_{\min}(\varepsilon) = \varepsilon$. ☐

Therefore, compositions of the border-preserving mappings with the string morphisms result in monotonic lattice mappings. Hence, the image lattices can be analysed with the same algorithms as the lattice described in Section III.

Moreover, if a string morphism maps elements of Σ into elements of Σ^* (i.e. acts in the same alphabet), then by the Knaster–Tarski theorem [14] the set of its fixpoints forms a complete sublattice of the lattice $\mathbf{WL_0}$, and no additional construction is required to track the properties captured by the morphism. Such a sublattice is shown in Fig. 4 in framed nodes and thick edges.

Tracing both the properties given in the lattice $\mathbf{WL_0}$ and in its sublattices wrt the string morphisms can be useful, for example, in the following analyses.

First, we can obtain a simple length analysis of strings, tracking constant values of string lengths. Indeed, the morphism $h_{\Sigma \to a}$ defined as $\forall c \in \Sigma\big(h_{\Sigma \to a}(c) = a\big)$ maps any given string to the unary Peano number representing its length.

Second, we can obtain a symbol occurrence analysis. Occurrences of forbidden symbols (i.e. all symbols from a set $\Sigma' \subset \Sigma$) can be traced in the sublattice produced by the morphism $h_{\Sigma\setminus\Sigma' \to \varepsilon}$ defined as follows: $\forall c \in \Sigma'\big(h_{\Sigma\setminus\Sigma' \to \varepsilon}(c) = a\big) \ \&\ \forall c \in \Sigma \setminus \Sigma'\big(h_{\Sigma\setminus\Sigma' \to \varepsilon}(c) = \varepsilon\big)$.

Third, simple string classification wrt the letter sets that are contained in the strings may be done. Namely, given $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \ldots \Sigma_k$, where $\forall i, j(i \neq j \Rightarrow \Sigma_i \cap \Sigma_j = \varnothing)$, the morphism $h_{\Sigma_1,\ldots,\Sigma_k}$ mapping any symbol from Σ_i to a single letter a_i produces a sublattice capturing abstract properties "to contain only the letters belonging to the set Σ_i".

VI. Related Works and Discussion

Practical string analysis tools [4], [8], [10], [23] tend to apply combinations of string domains. A natural abstraction of string sets may be expressed in terms of the regular language **RL** [6], [7], [9]. The **RL**-based abstract domains are easily defined given the union and intersection operations, however, the set of all regular languages cannot be directly used as an abstract domain. The main two problems to be solved when using the **RL**-based domains are listed below.

First, the lattice based on **RL** abstract domain admits infinite ascending chains. Thus, the straightforward usage of the regular expressions as elements of the lattice results in non-termination of the abstract interpretation.

Second, when the first problem is addressed via widening, infinite descending chains can still remain. The termination issue of the abstract interpretation relies only on the upper semi-lattice completeness. But the existence of such descending chains may indicate that the convergence speed is not uniformly bounded wrt the program to be analysed. For example, if the two strings start with the same common prefix

ξ, and ξ is long enough, then their widening to $\xi.*$, where $.*$ defines an arbitrary string, (as defined in the paper [6]) may result in $|\xi|$ iterations computing the upper bound, if the loop dropping the first letter of a word is analysed. The widening defined in the book [7] shares this feature as well.

The paper [9] discussed the following four abstract string domains often used in practical string analysis.

The first is an abstract domain with values Eq_{ξ}. This domain is included in $\mathbf{WL_0}$ presented in this paper.

The second is an abstract domain with values tracking the string lengths. Its simplest version can be modelled in a sublattice of $\mathbf{WL_0}$ by means of the morphism $h_{\Sigma\to a}$. Versions of the string length domain involving more complex length properties are independent from $\mathbf{WL_0}$ and can be used in the direct product with $\mathbf{WL_0}$ in order to improve preciseness of the analysis [24].

The third is an abstract domain with values representing predicates "string X contains a letter a". If the known set of the values $\Sigma' \subset \Sigma$ is tracked, then this domain is embedded in $\mathbf{WL_0}$ as a set of sublattices, by means of providing the set of string morphisms $h_{\Sigma\setminus\{c\}\to\varepsilon}$ mapping a chosen $c \in \Sigma'$ into itself, and all the other letters from Σ to ε.

The fourth is an abstract domain with values representing prefix predicates "string X starts with ξ", and the corresponding domain of suffix predicates. This domain contains infinite descending chains, since if $\xi_1\xi_2$ is a prefix of X, then ξ_1 (including $\xi_1 = \varepsilon$) is also a prefix of X.

The authors of the paper [10] use an abstract string domain separating unknown strings into numeric, non-numeric and special strings reflecting key words of JS syntax. Although the string domain is hard-coded, the JSAI tool makes use of configurable sensitivity in the trace analysis, thus allowing a user to redefine the tracked breakpoints. Thus, the idea presented in this paper to make the string analysis configurable by constructing the tokenizer mapping can be considered as an attempt to make the string-specified domain more configurable.

The works combining expressiveness of the word equation languages and regular languages emerged in program verification for finding loop invariants and pruning unreachable computation branches, see e.g. the paper [23]. There the straight-line fragment of the word equation language is considered, i.e. the variables in an equation cannot occur more than once. Nevertheless, such a fragment can still express some of language properties that are non-expressible by means of the multi-track finite automata [7].

The fresh work [25] reasons on so-called chain-free word equations, in which the variable dependences are bounded. The decision procedures for the existential theory of the chain-free equations together with regular constraints are given.

A. *Complexity of operations in* $\mathbf{WL_0}$

Several operations presented in this paper depend on finding either a primitive root of a string or its maximal suffix and prefix being a power of the same primitive word, given concatenation, string subtraction, and replacement operators. This task can be efficiently solved, e.g. by means of suffix arrays and LSP arrays, hence, the resulting complexity of the operations over abstract values x and y can be estimated as $O\big((|x| + |y|)\log(|x| + |y|)\big)$, where $|\,\mathrm{Eq}_{\omega}\,| = |\omega|$, and $|\,\mathrm{Cnj}_{\xi_1,\xi_2}\,| = |\xi_1| + |\xi_2|$.

VII. Conclusion and Future Work

In this paper, we have presented a first attempt to use the word equations as a basic language for constructing a string abstract domain. We have introduced the first, quantifier-free, layer of the resulting lattice $\mathbf{WL_0}$, together with the interpretation of the usual string processing operations in the domain based on the lattice $\mathbf{WL_0}$.

Extending the lattice with existential predicates (i.e. equations involving at least two variables) is an interesting and non-trivial task. A simplest choice of the existential two-variable predicates is to take one-variable patterns, i.e. equations of the form $X = \xi_1 Y \xi_2 \ldots Y \xi_{n+1}$. However, if we consider arbitrary one-variable patterns, the domain will include unbounded descending chains, e.g. $X = a_1 \ldots a_n Y, \ldots, X = a_1 Y$, which can make abstract interpretation too slow. Moreover, some predicates of this form can violate the upper semilattice condition. E.g. given an abstract value of the form $\mathrm{Cnj}_{\xi_1,\xi_2}$ if the predicate $\exists Y(X = YY)$ is also introduced as an abstract value, then $\mathrm{Eq}_{aa} \vee \mathrm{Eq}_{\varepsilon}$ becomes undefined, because we cannot choose the least element from $\mathrm{Cnj}_{\varepsilon,a}$ and $\exists Y(X = YY)$ unless we introduce additional lattice layers beyond the layer consisting of elementary equations.

Nevertheless, the patterns are the interesting and expressible language to be considered as a closest development of $\mathbf{WL_0}$.

Definitely there are other possibilities of the lattice enhancing, e.g. with the balanced two-variable equations. An equation is called balanced if multisets of the terms in its left- and right-sides coincide. For example, the predicates of the form $\exists Y(X\omega_1\omega_3\omega_2 Y = Y\omega_2\omega_3\omega_1 X)$, as well as the patterns, are basic in languages of two-variable equations, as shown in the paper [26]. I.e. an infinite language of any two-variable equation wrt variable X either contains a pattern or words satisfying the predicate $\exists Y(X\omega_1\omega_3\omega_2 Y = Y\omega_2\omega_3\omega_1 X)$, maybe intersected with a language of $\mathrm{Cnj}_{\xi_1,\xi_2}$. This approach has several benefits. First, the balanced two-variable equations as the abstract values can capture non-trivial properties of one-variable solution set projections, e.g. the X-solution set of the equation $XaYYb = YaXbY$ is $\big\{(b^n a)^m b^n \mid m, n \in \mathbb{N}\big\}$, describing a non-regular property of the X value. Second, the two-variable equations are able to express relations between concrete values over that the given variables can range. E.g., the solution set of the equation $XaY = YaX$ is $\big\{((\omega a)^*\omega, (\omega a)^*\omega) \mid \omega \in \Sigma^*\big\}$, where ω is a word parameter. While both X- and Y-projections of the solution set are trivial, the whole set indicates that X and Y values consist of repetitions of the same substring, separated with the letter a. In the case a solution-set description includes word parameters a static analysis may track not only known but also unknown repeated substrings in the data to be analysed.

If we do not restrict the equations with k variables, then construction of joins and meets becomes even harder, since, e.g. the pattern language inclusion is undecidable [27].

Acknowledgements

The author of the paper thanks Andrei Nemytykh for the support and many fruitful discussions on the draft of this paper, Denis Fokin for posing the problem of investigating string domain lattices; and anonymous referees for their valuable comments on the paper.

The research has been partially supported by Huawei Technologies Co. Ltd., and by Russian Academy of Sciences, research project №122012700089-0.

References

[1] J. D. Day, V. Ganesh, N. Grewal, and F. Manea, "Formal languages via theories over strings," *CoRR*, vol. abs/2205.00475, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.00475

[2] V. G. Durnev, "Undecidability of a simple fragment of a positive theory with a single constant for a free semigroup of rank 2, (in Russian)," *Matem. Zametki*, vol. 67, pp. 191–200, 2000. [Online]. Available: https://doi.org/10.4213/mzm827

[3] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 307–321. [Online]. Available: https://doi.org/10.1007/978-3-642-00768-2_27

[4] A. Reynolds, A. Nötzli, C. W. Barrett, and C. Tinelli, "Reductions for strings and regular expressions revisited," in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 225–235. [Online]. Available: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30

[5] B. Eriksson, A. Stjerna, R. D. Masellis, P. Rümmer, and A. Sabelfeld, "Black Ostrich: Web application scanning with string solvers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 549–563. [Online]. Available: https://doi.org/10.1145/3576915.3616582

[6] T. Choi, O. Lee, H. Kim, and K. Doh, "A practical string analyzer by the widening approach," in *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, ser. Lecture Notes in Computer Science, N. Kobayashi, Ed., vol. 4279. Springer, 2006, pp. 374–388. [Online]. Available: https://doi.org/10.1007/11924661_23

[7] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin, *String Analysis for Software Verification and Security*. Springer, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-68670-7

[8] B. Loring, D. Mitchell, and J. Kinder, "Sound regular expression semantics for dynamic symbolic execution of JavaScript," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 425–438. [Online]. Available: https://doi.org/10.1145/3314221.3314645

[9] V. Arceri, M. Olliaro, A. Cortesi, and I. Mastroeni, "Completeness of abstract domains for string analysis of JavaScript programs," in *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings*, ser. Lecture Notes in Computer Science, R. M. Hierons and M. Mosbah, Eds., vol. 11884. Springer, 2019, pp. 255–272. [Online]. Available: https://doi.org/10.1007/978-3-030-32505-3_15

[10] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: a static analysis platform for JavaScript," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 121–132. [Online]. Available: https://doi.org/10.1145/2635868.2635904

[11] F. Logozzo and H. Venter, "RATA: rapid atomic type analysis by abstract interpretation - application to JavaScript optimization," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 66–83. [Online]. Available: https://doi.org/10.1007/978-3-642-11970-5_5

[12] ECMA. (2023) ECMAScript specification. [Online]. Available: https://ecma-international.org/publications-and-standards/standards/ecma-262/

[13] G. Birkhoff and C. Bartee, *Modern Applied Algebra*, 1970.

[14] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific J. Math.*, vol. 5, no. 2, pp. 285–309, 1955. [Online]. Available: https://doi.org/10.2140/pjm.1955.5.285

[15] G. S. Makanin, "The problem of solvability of equations in a free semigroup," *Mat. Sb. (N.S.)*, vol. 103(145), pp. 147–236, 1977. [Online]. Available: https://doi.org/10.1070/SM1977v032n02ABEH002376

[16] J. Karhumäki, F. Mignosi, and W. Plandowski, "The expressibility of languages and relations by word equations," *J. ACM*, vol. 47, no. 3, pp. 483–505, May 2000. [Online]. Available: http://doi.acm.org/10.1145/337244.337255

[17] J. I. Hmelevskij, "Equations in a free semigroup, (in Russian)," *Trudy Mat. Inst. Steklov*, vol. 107, p. 286, 1971. [Online]. Available: https://www.mathnet.ru/rus/tm2975

[18] C. Choffrut and J. Karhumäki, "Combinatorics of words," in *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*, G. Rozenberg and A. Salomaa, Eds. Springer, 1997, pp. 329–438. [Online]. Available: https://doi.org/10.1007/978-3-642-59136-5_6

[19] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973

[20] A. Saarela, "Systems of word equations, polynomials and linear algebra: A new approach," *Eur. J. Comb.*, vol. 47, pp. 1–14, 2015. [Online]. Available: https://doi.org/10.1016/j.ejc.2015.01.005

[21] N. K. Kosovskij, "Some properties of solutions to equations in a free semigroup (in Russian)," *Zap. Nauch. Sem. LOMI*, vol. 32, pp. 21–28, 1972. [Online]. Available: https://www.mathnet.ru/rus/znsl2560

[22] D. Nowotka and A. Saarela, "An optimal bound on the solution sets of one-variable word equations and its consequences," *SIAM J. Comput.*, vol. 51, no. 1, pp. 1–18, 2022. [Online]. Available: https://doi.org/10.1137/20m1310448

[23] T. Chen, A. Flores-Lamas, M. Hague, Z. Han, D. Hu, S. Kan, A. W. Lin, P. Rümmer, and Z. Wu, "Solving string constraints with regex-dependent functions through transducers with priorities and variables," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–31, 2022. [Online]. Available: https://doi.org/10.1145/3498707

[24] T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, and Z. Wu, "A decision procedure for path feasibility of string manipulating programs with integer data type," in *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. V. Hung and O. Sokolsky, Eds., vol. 12302. Springer, 2020, pp. 325–342. [Online]. Available: https://doi.org/10.1007/978-3-030-59152-6_18

[25] D. Chocholatý, T. Fiedor, V. Havlena, L. Holík, M. Hruška, O. Lengál, and J. Síč, "Mata: A fast and simple finite automata library," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 130–151. [Online]. Available: https://doi.org/10.1007/978-3-031-57249-4_7

[26] L. Ilie and W. Plandowski, "Two-variable word equations," *RAIRO Theor. Informatics Appl.*, vol. 34, no. 6, pp. 467–501, 2000. [Online]. Available: https://doi.org/10.1051/ita:2000126

[27] T. Jiang, A. Salomaa, K. Salomaa, and S. Yu, "Inclusion is undecidable for pattern languages," in *Automata, Languages and Programming*, A. Lingas, R. Karlsson, and S. Carlsson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 301–312. [Online]. Available: https://dl.acm.org/doi/10.5555/646247.684876

Appendix

Computation of $x - y$

A word ξ is said to be a fractional power of τ, if $\exists\omega_1, \omega_2, k\big(k \in \mathbb{N} \ \&\ \tau = \omega_1\omega_2 \ \&\ \xi = \tau^k\omega_1\big)$. The fractional power of τ in ξ is computed as $k + \frac{|\omega_1|}{|\tau|}$ [18]. E.g. $abaabaa = (aba)^{2\frac{1}{3}}$. Hence, words satisfying any predicate $\mathrm{Cnj}_{\omega_1,\omega_2}$ are fractional powers of $\omega_1\omega_2$ with the non-integer fractional part consisting of ω_1 (the fractional part is non-empty if $|\omega_1| \neq 0$). We recall that the operation $a(x)$ takes a concrete value of a predicate x.

Below the definition of the prefix subtraction operation in $\mathbf{WL_0}$ is repeated.

$$x - y = \begin{cases} \mathrm{Eq}_\tau, \text{ if } x = \mathrm{Eq}_{\xi\tau} \ \&\ y = \mathrm{Eq}_\xi; \\ \text{error, if } x = \mathrm{Eq}_\tau \ \&\ y = \mathrm{Eq}_\xi \ \&\ \forall\tau'(\tau \neq \xi\tau'); \\ \text{error, if } x = \mathrm{Cnj}_{\tau_1,\tau_2} \ \&\ y = \mathrm{Eq}_\xi \\ \qquad \&\ \neg(y \leq \mathrm{Cnj}_{\xi_1,\xi_2}), \text{ where } \tau_1\tau_2 = \xi_1\xi_2; \\ x, \text{ if } y = \mathrm{Cnj}_{\varepsilon,\xi} \text{ and} \\ \quad \text{either} \quad x = \mathrm{Eq}_\tau \quad \text{and} \quad \forall\tau'\big(\tau \neq \xi\tau'\big), \\ \quad \text{or } x = \mathrm{Cnj}_{\tau_1,\tau_2} \text{ and } \forall\tau', k\big((\tau_1\tau_2)^k\tau_1 \neq \xi\tau'\big); \\ \top, \text{ if } y = \mathrm{Cnj}_{\xi_1,\xi_2} \text{ and } x - y \text{ can satisfy at least} \\ \quad \text{two different predicates of the form } \mathrm{Cnj}_{\omega_1,\omega_2}; \\ \mathrm{Cnj}_{\tau_{1,2},\tau_2\tau_{1,1}}, \text{ if } x \leq \mathrm{Cnj}_{\tau_1,\tau_2} \text{ and } y \leq \mathrm{Cnj}_{\xi_1,\xi_2} \\ \qquad\qquad \text{and } \exists k, k', \tau_{1,1}, \tau_{1,2} \\ \quad \big((\tau_1\tau_2)^k\tau_{1,1} = (\xi_1\xi_2)^{k'}\xi_1 \ \&\ \tau_1 = \tau_{1,1}\tau_{1,2}\big); \\ \mathrm{Cnj}_{\tau_{2,2}\tau_1,\tau_{2,1}}, \text{ if } x \leq \mathrm{Cnj}_{\tau_1,\tau_2} \text{ and } y \leq \mathrm{Cnj}_{\xi_1,\xi_2} \\ \qquad\qquad \text{and } \exists k, k', \tau_{2,1}, \tau_{2,2} \\ \quad \big((\tau_1\tau_2)^k\tau_1\tau_{2,1} = (\xi_1\xi_2)^{k'}\xi_1 \ \&\ \tau_2 = \tau_{2,1}\tau_{2,2}\big); \\ \top, \text{ otherwise.} \end{cases}$$

The first three cases of the definition are self-explanatory; now we consider the case returning x. Given $y = \mathrm{Cnj}_{\varepsilon,\xi}$, if any concrete value of $a(x)$ does not start with ξ, the only value of $a(y)$ that can be subtracted from $a(x)$ is ε, i.e. ξ^0.

In the following cases, we treat the predicate $x = \mathrm{Eq}_\tau$ uniformly with $\mathrm{Cnj}_{\tau_1,\tau_2}$, making use of the fact that any τ can be represented as $(\tau_1\tau_2)^m\tau_1$, where $\tau_1 = \varepsilon$, $\tau_2 = \rho(\tau)$.

Let $x = \mathrm{Cnj}_{\tau_1,\tau_2}$ or $x = \mathrm{Eq}_{(\tau_1\tau_2)^m\tau_1}$, $y = \mathrm{Cnj}_{\xi_1,\xi_2}$, and let $a(x)$ start with $a(y)$. Then $\exists k_1\big(a(x) = (\tau_1\tau_2)^{k_1}\tau_1\big)$; $\exists k_2\big(a(y) = (\xi_1\xi_2)^{k_2}\xi_1\big)$; $\exists\tau'\big(a(x) = a(y)\tau'\big)$.

If both ξ_1 and $\xi_1\xi_2\xi_1$ are fractional powers of $\tau_1\tau_2$ and the fractional parts of $\tau_1\tau_2$ in ξ_1 and $\xi_1\xi_2\xi_1$ do not coincide, then the abstract value of $a(x) - a(y)$ cannot be determined, and the value $\top$ is returned by the computation of $x - y$ (shown in the fifth case of the definition).

In the remaining cases below the fifth (the one returning $\top$), we assume that either ξ_1 and $\xi_1\xi_2\xi_1$ are powers of $\tau_1\tau_2$ and the fractional parts of $\tau_1\tau_2$ in ξ_1 and $\xi_1\xi_2\xi_1$ coincide, or that ξ_1 is a power of $\tau_1\tau_2$, while $\xi_1\xi_2\xi_1$ is not.

If the string $a(y)$ ends inside the string τ_1, then $\tau_1 = \tau_{1,1}\tau_{1,2}$, and $\exists k_3\big(\tau' = (\tau_{1,2}\tau_2\tau_{1,1})^{k_3}\tau_{1,2}\big)$. Hence, τ' satisfies the predicate $\mathrm{Cnj}_{\tau_{1,2},\tau_2\tau_{1,1}}$ (the sixth case of the definition).

If the string $a(y)$ ends inside the string τ_2, then $\tau_2 = \tau_{2,1}\tau_{2,2}$, and $\exists k_3\big(\tau' = (\tau_{2,2}\tau_1\tau_{2,1})^{k_3}\tau_{2,2}\tau_1\big)$. Hence, τ' satisfies the predicate $\mathrm{Cnj}_{\tau_{2,2}\tau_1,\tau_{2,1}}$ (the seventh case of the definition).

Note that the order of the right-hand sides of the interpretation rule for $x - y$ guarantees that if $a(y)$ satisfies the predicate $\mathrm{Cnj}_{\tau_1,\tau_2}$ then the resulting abstract value is $\mathrm{Cnj}_{\varepsilon,\tau_2\tau_1}$, but not $\mathrm{Cnj}_{\tau_2\tau_1,\varepsilon}$. These two predicates are equivalent, but only the former is consistent with the definition of the abstract values in $\mathbf{WL_0}$.

Formally Verifying Deep Reinforcement Learning Controllers with Lyapunov Barrier Certificates

Udayan Mandal[1], Guy Amir[2], Haoze Wu[1], Ieva Daukantas[3], Fletcher Lee Newell[1], Umberto J. Ravaioli[4], Baoluo Meng[5], Michael Durling[5], Milan Ganai[1], Tobey Shim[1], Guy Katz[2], and Clark Barrett[1]
[1]Stanford University, Stanford, United States, {udayanm, haozewu, flnewell, mganai, tshim24, barrett}@stanford.edu
[2]The Hebrew University of Jerusalem, Jerusalem, Israel, {guyam, guykatz}@cs.huji.ac.il
[3]IT University of Copenhagen, Copenhagen, Denmark, daukantas@itu.dk
[4]Google, Mountain View, United States, uravaioli@google.com
[5]GE Aerospace Research, Niskayuna, United States, {baoluo.meng, durling}@ge.com

***Abstract*—Deep reinforcement learning (DRL) is a powerful machine learning paradigm for generating agents that control autonomous systems. However, the "black box" nature of DRL agents limits their deployment in real-world safety-critical applications. A promising approach for providing strong guarantees on an agent's behavior is to use *Neural Lyapunov Barrier* (NLB) certificates, which are learned functions over the system whose properties indirectly imply that an agent behaves as desired. However, NLB-based certificates are typically difficult to learn and even more difficult to verify, especially for complex systems. In this work, we present a novel method for training and verifying NLB-based certificates for discrete-time systems. Specifically, we introduce a technique for certificate *composition*, which simplifies the verification of highly-complex systems by strategically designing a sequence of certificates. When jointly verified with neural network verification engines, these certificates provide a formal guarantee that a DRL agent both achieves its goals and avoids unsafe behavior. Furthermore, we introduce a technique for certificate *filtering*, which significantly simplifies the process of producing formally verified certificates. We demonstrate the merits of our approach with a case study on providing safety and liveness guarantees for a DRL-controlled spacecraft.**

I. Introduction

In recent years, deep reinforcement learning (DRL) has achieved unprecedented results in multiple domains, including game playing, robotic control, protein folding, and many more [22], [49], [65], [72]. However, such models have an opaque decision-making process, making it highly challenging to determine whether a DRL-based system will *always* behave correctly. This is especially concerning for safety-critical domains (e.g., autonomous vehicles), in which even a single mistake can have dire consequences and risk human lives. This drawback limits the incorporation of DRL in real-world safety-critical systems.

The formal methods community has responded to this challenge by developing automated reasoning approaches for *proving* that a DRL-based controller behaves correctly [62]. These efforts rely in part on specialized DNN verification engines (a.k.a. *DNN verifiers*), which adapt techniques from other domains such as satisfiability modulo theories, abstract interpretation, mixed integer linear programming, and convex optimization [55], [56], [67], [87]. DNN verifiers take as input a DNN and a specification of the desired property and produce either a *proof* that the property always holds, or a *counterexample* demonstrating a case where the property does not hold. While the scalability of DNN verifiers has improved dramatically in the past decade [20], they struggle when applied to *reactive* (e.g., DRL-based) systems with temporal properties which require reasoning about interactions with the environment over time. This is because a naive approach for reasoning about time requires the involved DNN to be *unrolled* (i.e., a copy made for each time step), greatly increasing the complexity of the verification task.

On the other hand, for dynamical systems, a traditional approach for guaranteeing temporal properties has been to use control certificates such as Lyapunov Barrier functions [63]. Unfortunately, standard approaches for constructing these functions are not easily applicable to DRL-based dynamical systems. Recently, however, techniques have been developed for *learning* control certificates. We call these *Neural Lyapunov Barrier* (NLB) certificates [33]. Although NLB-based approaches have been shown to work for simple, toy examples, these certificates have been, thus far, difficult to learn and verify for real-world systems, which often involve large state spaces with complex dynamics.

In this work, we present a novel framework for training and formally verifying NLB-based certificates. Our framework can verify both *liveness* and *safety* properties of interest, providing *reach-while-avoid* (RWA) guarantees. We use off-the-shelf DNN verifiers and introduce a set of novel techniques to improve scalability, including certificate *filtering* and *composition*.

We demonstrate our approach with a case study targeting a specific challenge problem, in which the goal is to verify a DRL-based spacecraft controller [78]. We show that our framework is able to generate verified NLB-based RWA certificates for a range of complex properties. These include liveness properties (e.g., *will the spacecraft eventually reach its destination?*) and complex non-linear safety properties (e.g., *the spacecraft will never violate a non-linear velocity constraint*), both of which are challenging to verify using existing techniques.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_15

The rest of this paper is organized as follows: Sec. II gives an overview of relevant background material on property types, DNN verifiers, and NLB certificates. Related work is covered in Sec. III. In Sec. IV and V, we present our approach, and Sec. VI reports the results of our spacecraft case study.[1] Finally, Sec. VII concludes.

Note. Proofs and additional details can be found in an extended technical report [69].

II. Preliminaries

A. Property Types

This work focuses on DRL controllers that are invoked over discrete time steps. We consider both safety and liveness properties [5].

Safety. A safety property indicates that *a bad state is never reached*. More formally, let $\mathcal{X}$ be the set of system states, and let $\tau \subseteq \mathcal{X}^*$ be the set of possible system trajectories. The system satisfies a *safety* property P if and only if every state in every trajectory satisfies P:

$$\forall \alpha : \alpha \in \tau : (\forall x \in \alpha : x \vDash P) \tag{1}$$

A violation of a safety property is a finite trajectory ending in a "bad" state (i.e., a state in which P does not hold).

Liveness. A liveness property concerns the *eventual* behavior of a system (e.g., *a good state is eventually reached*). More formally, we say a *liveness* property P holds if and only if there exists a state x in every infinite trajectory where P holds. Letting τ^∞ be the set of infinite trajectories, we can formalize this as follows.

$$\forall \alpha : \alpha \in \tau^\infty : (\exists x \in \alpha : x \vDash P), \tag{2}$$

A violation of a liveness property is an infinite trajectory in which each state violates the property P.

B. DNNs, DNN Verification, and Dynamical Systems.

Deep Learning. Deep neural networks (DNNs) [43] consist of layers of neurons, each layer performing a (typically non-linear) transformation of its input. This work focuses on deep reinforcement learning (DRL), a popular paradigm in which a DNN is trained to realize a *policy*, i.e., a mapping from states (the DNN's inputs) to actions (the DNN's outputs), which is used to control a reactive system. For more details on DRL, we refer to [64].

DNN Verification. Given (i) a trained DNN (e.g., a DRL agent) N; (ii) a precondition P on the DNN's inputs, limiting the input assignments; and (iii) a postcondition Q on the DNN's output, the goal of DNN verification is to determine whether the property $P(x) \rightarrow Q(N(x))$ holds for any neural network input x. In many DNN verifiers (a.k.a., *verification engines*), this task is equivalently reduced to determining the satisfiability of the formula $P(x) \wedge \neg Q(N(x))$. If the formula is satisfiable (`SAT`), then there is an input that satisfies the pre-condition and violates the post-condition, which means the property is violated. On the other hand, if the formula is unsatisfiable (`UNSAT`), then the property holds. It has been shown [55] that verification of piecewise-linear DNNs is NP-complete.

[1]Code for reproducing the experiments is available at: github.com/NeuralNetworkVerification/artifact-fmcad24-docking.

Discrete Time-Step Dynamical Systems. We focus on dynamical systems that operate in a discrete time-step setting. More formally, these are systems whose trajectories satisfy the equation:

$$x_{t+1} = f(x_t, u_t), \tag{3}$$

where f is a *transition function* that takes as inputs the current state $x_t \in \mathcal{X}$ and a control input $u_t \in \mathcal{U}$ and produces the next state x_{t+1}. These systems are controlled using a feedback control policy $\pi : \mathcal{X} \rightarrow \mathcal{U}$ which, given a state $x \in \mathcal{X}$ produces control input $u = \pi(x)$. In our setting, the controller π is realized by a DNN trained using DRL. DRL-based controllers are potentially useful in many real-world settings, due to their expressivity and their ability to generalize to complex environments [85].

C. Control Lyapunov Barrier Functions

The problem of verifying a liveness or safety property over a dynamical system with a given control policy can be reduced to the task of identifying a certificate function $V : \mathcal{X} \mapsto \mathbb{R}$, whose input-output relation satisfies a particular set of constraints that imply the property. There are two fundamental types of certificate functions.

Lyapunov Functions. A Lyapunov function (a.k.a. *Control Lyapunov function*) represents the energy level at the current state: as time progresses, energy is dissipated until the system reaches the zero-energy equilibrium point [48]. Hence, such functions are typically used to provide *asymptotic stability*, i.e., adherence to a desired liveness property, or the eventual convergence of the system to some goal state. Such guarantees can be afforded by learning a function that (i) reaches a 0 value at equilibrium, (ii) is strictly positive everywhere else; and (iii) either monotonically decreases [25], [26] or decreases by a particular constant [40] with each time step.

Barrier Functions. Barrier functions [8], a.k.a. *Control Barrier Functions*, are also energy-based certificates. However, these functions are typically used for verifying safety properties. Barrier functions enforce that a system will never enter an unsafe region in the state space. This is done by assigning unsafe states a function value above some threshold and then verifying that barrier function never crosses this threshold [7], [16], [90].

Control Lyapunov Barrier Functions. In many real-world settings, it can be useful to verify both liveness properties and safety properties. In such cases, a *Control Lyapunov Barrier Function* (CLBF) can be used, which combines the properties of both Control Barrier functions and Lyapunov functions. CLBFs can provide rigorous guarantees w.r.t. a wide variety of temporal properties, including the general setting of reach-while-avoid tasks [37], which we describe next.

Reach-while-Avoid Tasks. In a *reach-while-avoid* (RWA) task, we must find a controller π for a dynamical system such that all trajectories $\{x_1, x_2...\}$ produced by this controller (i) do not include unsafe ("bad") states; and (ii) eventually reach a goal state. More formally the problem can be defined as follows:

Definition 1 (Reach-while-Avoid Task).
Input: *A dynamical system with a set of initial states $\mathcal{X}_I \subseteq \mathcal{X}$, a set of goal states $\mathcal{X}_G \subseteq \mathcal{X}$, and a set of unsafe states $\mathcal{X}_U \subseteq \mathcal{X}$, where $\mathcal{X}_I \cap \mathcal{X}_U = \varnothing$ and $\mathcal{X}_G \cap \mathcal{X}_U = \varnothing$*
Output: *A controller π such that for every trajectory $\tau = \{x_1, x_2...\}$ satisfying $x_1 \in \mathcal{X}_I$:*
1) ***Reach:*** $\exists\, t \in \mathbb{N}.\, x_t \in \mathcal{X}_G$
2) ***Avoid:*** $\forall\, t \in \mathbb{N}.\, x_t \notin \mathcal{X}_U$

III. Related Work

A. Control Certificates

Control certificate-based approaches form a popular and effective class of methods for providing guarantees about complex dynamical systems in diverse application areas including robotics [33], energy management [52], and biomedical systems [35]. Control Lyapunov functions are certificates for system stability, and the closely-related control Barrier functions are certificates for safety. While such Lyapunov-based certificates have been proposed over a century ago [66], their main drawback lies in their computational intractability [42]. As a result, practitioners have mainly relied on unscalable methods for constructing certificates, such as manual design for domain-dependent certificate functions [24], [27], sum-of-squares approaches restricted to polynomial systems [53], [68], and quadratic programming [63].

1) Formal Verification of Neural Certificates: Recent methods have leveraged neural networks as verifiable models of these control certificates, forming a class of *neural certificate* approaches [33]. For a fixed controller, [80] distills the problem into solving binary classification with neural networks, but the method is limited to polynomial systems and only obtains a region of attraction, making it incompatible with most RWA problems, which have a predefined goal region.

In [2], [4], SMT solvers are employed to check whether a certificate for a specific controller satisfies the Lyapunov conditions and, if not, to return counterexamples which can be used to retrain the neural certificate. A similar approach can be used for Barrier conditions [73]. In [1], [36], the Fossil tool is introduced, which combines these methods. In [3], Fossil is used to generate training examples for barrier certificates which are used to construct overapproximations of safe reach sets. However, these methods require verifying all constraints in the certificate for the entirety of the relevant state space — a task which can be computationally prohibitive (as we show in Section VI).

In [26], a Neural Lyapunov Control (NLC) framework is proposed, which jointly learns the Lyapunov certificate and the controller. The algorithm iteratively calls the dReal SMT solver [41] to generate counterexamples and retrain both the neural certificate and the control policy. Various extensions and applications followed: [45] addresses algorithmic problems in NLC; [46] automates the design of passive fault-tolerant control laws using NLC; [97] extends NLC to unknown nonlinear systems; [88] extends NLC to discrete-time systems; [82] verifies single hidden-layer ReLU neural certificates with enumeration [77] and linear programming; and [96] develops a framework for Barrier functions when there is an existing nominal controller. However, these methods do not consider the more general reach-avoid problem.

2) Data-driven Neural Certificates: To improve scalability, a recent line of research proposes learning certificates and controllers from online and/or offline data without additional formal verification [33], following the intuition that, with increasing data, the number of violations in the trained certificate will tend toward zero [19]. [25], [40] learn Lyapunov certificates for stabilization control, and [75], [76], [86], [95] synthesize neural Barrier functions in various settings like multi-agent control, neural radiance field [71] imagery, and pedestrian avoidance. These methods (by design) cannot provide rigorous guarantees on the validity of their learned certificates.

B. Reach-Avoid methods

Solutions for tasks requiring the simultaneous verification of both liveness and safety properties, of which the RWA task is a common example, have also relied on control theoretic principles. [34] learns a combined Lyapunov and Barrier certificate to construct controllers with stabilization and safety guarantees. The Hamilton-Jacobi (HJ) reachability-based method (a verification method for ensuring optimal control performance and safety in dynamical systems [15]) has also been used to solve reach-avoid problems [38], [51], [84]. Safe reinforcement learning is closely related to reach-avoid: the goal is to maximize cumulative rewards while minimizing costs along a trajectory [21], and it has been solved with both Lyapunov/Barrier methods [28], [91] and HJ reachability methods [39], [94]. As mentioned, scalability is a crucial challenge in this context. The next section describes our approach for addressing this challenge.

IV. Reach-While-Avoid Certificates

In this section, we present our approach for scalably creating verified NLB certificates. We first describe reach-while-avoid (RWA) certificates, a popular class of existing NLB-based certificates. We next present an extension called *Filtered* RWA certificates, which significantly simplifies the learning task and enables efficient training of certificates for complex properties. We then present a *compositional* certification approach, which independently trains a series of certificates that can be jointly verified to handle even larger state spaces.

A. RWA certificates

A function $V : \mathcal{X} \mapsto \mathbb{R}$ is an RWA certificate for the Reach-Avoid task in Definition 1 if, for some $\alpha > \beta$ and $\epsilon > 0$, it satisfies the following constraints.[2]

$$\forall\, x \in \mathcal{X}_I. \qquad V(x) \leq \beta \tag{4}$$

$$\forall\, x \in \mathcal{X} \setminus \mathcal{X}_G. \quad V(x) \leq \beta \rightarrow V(x) - V(f(x, \pi(x))) \geq \epsilon \tag{5}$$

$$\forall\, x \in \mathcal{X}_U. \qquad V(x) \geq \alpha \tag{6}$$

Any tuple of values $(\alpha, \beta, \epsilon)$ for which these conditions hold is called a *witness for* the certificate. RWA certificates provide the following guarantees.[3]

Lemma 1. *If V is an RWA certificate for a dynamical system with witness $(\alpha, \beta, \epsilon)$, and V has a lower bound,[4] then for every infinite trajectory τ starting from a state $x \in \mathcal{X} \setminus \mathcal{X}_G$ such that $V(x) \leq \beta$, τ will eventually contain a state in $\mathcal{X}_G$ without ever passing through a state in $\mathcal{X}_U$.*

Intuitively, V partitions the state space into three regions:

- a *safe region* where the value of the certificate is at most β. This region includes the initial states $\mathcal{X}_I$ and any states reachable from $\mathcal{X}_I$. Furthermore, starting from any non-goal state in the safe region, the certificate function value should decrease by *at least* ϵ at each time step.
- an *unsafe region* where the value of the certificate is at least α. This region must include the unsafe states $\mathcal{X}_U$.
- an *intermediate region*, where the value of the certificate is strictly between β and α. States in this region are not unsafe but are also not reachable from $\mathcal{X}_I$. This can also be thought of as a "buffer" region that separates the safe region from the unsafe region. These states play a role in the compositional approach described below.

B. FRWA certificates

A *neural RWA* certificate is an RWA certificate realized by a DNN. Such a DNN can be trained by following the NLC approach [26], using the constraints (4)–(6) as training objectives. Because we are also interested in formally verifying these certificates, we would like to keep the DNNs (both the controller and the certificate) small so that verification remains tractable. We have observed that this can be challenging when the system and properties are non-trivial. To help address this, we introduce an improvement called *Filtered Reach-while-Avoid* (FRWA) certificates.

The idea behind FRWA is straightforward. Often, we can describe the goal and unsafe regions using simple predicates (or filters) on the state space. We pick constants c_1, c_2 such that $c_1 \leq \beta < \alpha \leq c_2$ and then hard-code the implementation of V so that $x \in \mathcal{X}_G \rightarrow V(x) = c_1$ and $x \in \mathcal{X}_U \rightarrow V(x) = c_2$. Note that the latter ensures that condition (6) holds by construction. Importantly, this not only makes the training task easier, but also reduces the number of queries required to formally verify the certificate. On the other hand, hard-coding the certificate value for inputs in $\mathcal{X}_G$ makes it easier to learn constraint (5). The reason for this is more nuanced. If we randomly initialize the certificate neural network, the certificate value for some states in $\mathcal{X}_G$ could start out larger than β, making it more difficult to satisfy constraint (5) for a point x where $V(x) \leq \beta$ and $f(x, \pi(x)) \in \mathcal{X}_G$. Fixing the certificate values for states in $\mathcal{X}_G$ to at most β (ideally, significantly below β) ensures that, at least for such points, condition (5) is easier to satisfy. In practice, FRWA certificates can be implemented by using a wrapper around a DNN which checks the two filters and only calls the DNN if they both fail. The practical effectiveness of FRWA certificates is demonstrated in Sec. VI.

FRWA Training. FRWA simplifies the certificate learning process, as now, only constraints 4 and 5 are relevant for training. We custom design the reinforcement learning training objective function as follows. Let $x_1, \ldots, x_N$ be the set of training points, and let $x'_i = f(x_i, \pi(x_i))$. We define:

$$O_s = c_s \sum_{i \mid x_i \in \mathcal{X}_I} \frac{\mathrm{ReLU}(\delta_1 + V(x_i) - \beta)}{\sum_{i \mid x_i \in \mathcal{X}_I} 1} \tag{7}$$

$$O_d = c_d \sum_{i \mid x_i \in \mathcal{X} \setminus (\mathcal{X}_U \cup \mathcal{X}_G), V(x_i) \leq \beta} \frac{\mathrm{ReLU}(\delta_2 + \epsilon + V(x'_i) - V(x_i))}{\sum_{i \mid x_i \in \mathcal{X} \setminus (\mathcal{X}_U \cup \mathcal{X}_G), V(x_i) \leq \beta} 1} \tag{8}$$

$$O = O_s + O_d \tag{9}$$

Eq. (7) penalizes deviations from constraint (4), and Eq. (8) penalizes deviations from constraint (5). We incorporate parameters $\delta_1 > 0$ and $\delta_2 > 0$, which can be used to tune how strongly the certificate over-approximates adherence to each constraint. Similarly, constants c_s and c_d can be used to tune the relative weight of the two objectives. The final training objective O in (9) is what the optimizer seeks to minimize, by using stochastic gradient descent (SGD) or other optimization techniques. We note that the FRWA certificates are trained in a self-supervised, non-RL setting.

FRWA Data Sampling. From the formulation above, we see that only data points in $(\mathcal{X} \setminus (\mathcal{X}_U \cup \mathcal{X}_G)) \cup \mathcal{X}_I$ affect the objectives, and thus, only these data points need to be sampled.

FRWA Verification. We use DNN verification tools to formally verify that conditions (4)-(6) hold for our certificates. Filtering introduces a slight complication. Recall that a FRWA certificate is implemented as a wrapper around a DNN, meaning that the DNN itself can behave arbitrarily when either $x \in \mathcal{X}_G$ or $x \in \mathcal{X}_U$. Fortunately, we can adjust the verification conditions for the DNN part of the certificate as follows.

Constraint (4) can be checked as is. The filtering does not affect this property. And it is easy to see that checking the property for the DNN does indeed ensure the property holds for the full certificate.

Constraint (6) need not be checked at all, as the filtered certificate ensures this condition by construction.

[2]These constraints are similar to the ones defined in prior work [37] but are specific to discrete time-step systems and instead place constraints on the set of unsafe states instead of a compact safe set.

[3]See [69] for a proof.

[4]This is always the case if the output of V is implemented using a finite representation such as floating-point arithmetic.

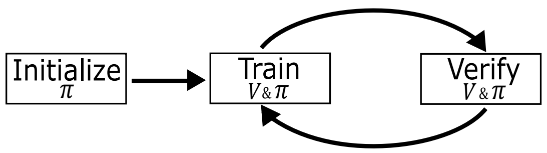


Fig. 1: The CEGIS Loop used to iteratively train and verify controller π and certificate V. Verification counterexamples are used to augment the training dataset.

Verification of constraint 5 is done by instead checking that:

$$\begin{aligned}&\forall\, x \in \mathcal{X} \setminus (\mathcal{X}_U \cup \mathcal{X}_G), x' \in \mathcal{X}.\\ &\quad (x' = f(x, \pi(x)) \wedge V(x) \le \beta) \rightarrow\\ &\qquad (V(x) - V(x') \ge \epsilon \vee (x' \in \mathcal{X}_G)) \wedge (x' \notin \mathcal{X}_U) \qquad (10)\end{aligned}$$

There are three main differences between (5) and (10). Since the filter ensures that $V(x) > \beta$ when $x \in \mathcal{X}_U$, we can safely exclude states in $\mathcal{X}_U$ from the check. Similarly, if the system ever transitions from a state x with $V(x) \le \beta$ to an unsafe state, the filter ensures that condition (5) is violated, so it suffices to check that $x' \notin \mathcal{X}_U$ to cover this case.

The last difference is a bit more subtle. Observe that (10) is trivially true if $x' \in \mathcal{X}_G$, meaning that if we transition to a goal state, we do not enforce (5). However, it is easy to see that Lemma 1 still holds with this relaxed condition: if every transition either reduces V by at least ϵ or reaches a goal state, then clearly, we must eventually reach a goal state.

C. CEGIS loop

We use a *counterexample-guided inductive synthesis* (CEGIS) loop, shown in Fig. 1, to obtain a fully verified controller and certificate. We first train an initial controller π. Then, at each CEGIS iteration, we jointly train V and π until a loss of 0 is obtained and then use a sound and complete DNN verifier (we use *Marabou* [58] in our experiments) to identify counterexamples. If the verifier identifies a counterexample violating constraints (4) or (5) (recall that constraint (6) is satisfied by construction), we sample points in the proximity of the counterexample and use these to augment the training data. By sampling multiple nearby points, we hope to influence the training to learn smooth behavior for a localized neighborhood instead of overfitting to a specific point. This process is repeated iteratively until no counterexamples are found, at which point we are guaranteed to have produced a fully verified controller and certificate.

V. Compositional Certificates

While filtering does improve the efficiency of both training and verification, the approach outlined above still suffers from scalability challenges, especially as the system complexity or state space covered by the controller increases. In this section, we introduce *compositional certificates*, which aim to aid scalability by training multiple controller-certificate pairs, each covering different parts of the state space. The certificates are compositional in the sense that a simple meta-controller can be designed to determine which controller-certificate pair to use when in a given state, and we can formally guarantee that the meta-controller satisfies the requirements of definition 1.

CRWA. Formally, a compositional RWA certificate (CRWA) for an RWA task is composed of n RWA certificates,[5] which we denote $V_0, \ldots, V_{n-1}$, with corresponding controllers, which we denote $\pi_0, \ldots, \pi_{n-1}$, with $n \ge 2$. Furthermore, each pair (V_i, π_i) must be an RWA certificate with some witness $(\alpha_i, \beta_i, \epsilon_i)$ for an RWA task whose dynamics are that of the main RWA task, but whose parameters are $(\mathcal{X}_I^i, \mathcal{X}_G^i, \mathcal{X}_U^i)$. These parameters must satisfy the following conditions:

(i) $\mathcal{X}_I^0 \subseteq \mathcal{X}_I$, $\mathcal{X}_G^0 = \mathcal{X}_G$, and $\mathcal{X}_U \subseteq \mathcal{X}_U^0 \subseteq \overline{(\mathcal{X}_I^0 \cup \mathcal{X}_G^0)}$, where $\overline{S}$ denotes the complement of the set S;
(ii) for $0 < i < n$, $\mathcal{X}_I^{i-1} \subseteq \mathcal{X}_I^i \subseteq \mathcal{X}_I$, $\mathcal{X}_G^i = \{x \in \overline{\mathcal{X}_U^{i-1}} \mid V_{i-1}(x) \le \beta_{i-1}\} \cup \mathcal{X}_G^0$, and $\mathcal{X}_U \subseteq \mathcal{X}_U^i \subseteq \mathcal{X}_U^{i-1}$;
(iii) either $\mathcal{X}_I^i \ne \mathcal{X}_I^{i-1}$ or $\mathcal{X}_U^i \ne \mathcal{X}_U^{i-1}$; and
(iv) $\mathcal{X}_I^{n-1} = \mathcal{X}_I$ and $\mathcal{X}_U^{n-1} = \mathcal{X}_U$.

Intuitively, the idea is as follows. We start with an initial controller capable of guiding the system from some *subset* of the initial states $\mathcal{X}_I$ to the original goal states $\mathcal{X}_G$ while avoiding some *superset* of the unsafe states $\mathcal{X}_U$. Then, for each subsequent controller, we ensure that it can guide the system either from a larger subset of the initial states $\mathcal{X}_I$ or while avoiding a smaller superset of the unsafe states $\mathcal{X}_U$, or both, to a new goal region consisting of the states considered safe by the previous controller, i.e., the states x for which $V(x) \le \beta$. For the final controller (controller $n-1$), the set of initial and unsafe states should coincide with those of the original RWA problem. Note that the algorithm does not say how to choose of $\mathcal{X}_I^i$ and $\mathcal{X}_U^i$ for $i < n-1$ other than to specify that these sets should be monotonically increasing and decreasing, respectively. Finding good heuristics for choosing these sets in the general case is a promising direction for future work.

The meta-controller behaves as follows. Given any starting state $x \in \mathcal{X}_I$, we first check if $x \in \mathcal{X}_G$. If so, we are done. Otherwise, we determine the smallest i for which $x \in \mathcal{X}_I^i$ and guide the system using π_i until a state in $\mathcal{X}_G^i$ is reached, which will occur in some finite number of steps because of the guarantees provided by V_i. At this point, we transition to π_{i-1}, and the process repeats until a state in $\mathcal{X}_G$ is reached.

The training and verification of a CRWA certificate is described in Alg. 1 and visualized in Fig. 2.

The following lemma captures the correctness of our approach.[6]

Lemma 2. *Given a CRWA certificate for an RWA task with parameters $\mathcal{X}_I$, $\mathcal{X}_G$, and $\mathcal{X}_U$, all trajectories guided by the meta-controller starting at any point in $\mathcal{X}_I$ will reach $\mathcal{X}_G$ in a finite number of steps while avoiding $\mathcal{X}_U$. In other words, a CRWA certificate provides a correct solution for the RWA task.*

CRWA Data Sampling. When training certificate V_i, it is important that the training dataset contains sufficient states

[5]Each controller in a CRWA can make use of the FRWA technique described above.
[6]See [69] for a proof.

Algorithm 1: CRWA Training and Verification

```
Input : $\mathcal{X}_I$, $\mathcal{X}_G$, $\mathcal{X}_U$
Output: $\pi_0, \dots, \pi_{n-1}, V_0, \dots, V_{n-1}$ for some $n$
1  $\mathcal{X}_G^0 \leftarrow \mathcal{X}_G$
2  Choose $\mathcal{X}_I^0 \subseteq \mathcal{X}_I$ and $\mathcal{X}_U^0 \supseteq \mathcal{X}_U$, with $\mathcal{X}_U^0 \subseteq \overline{(\mathcal{X}_I^0 \cup \mathcal{X}_G^0)}$
3  Choose $\alpha^0 > \beta^0$ and $\epsilon^0 > 0$
4  Train and verify controller $\pi_0$ and certificate $V_0$ with
     witness $(\alpha^0, \beta^0, \epsilon^0)$ for the RWA task corresponding
     to $\mathcal{X}_I^0, \mathcal{X}_G^0$, and $\mathcal{X}_U^0$ using, e.g., the approach shown
     in Fig. 1
5  $i \leftarrow 0$
6  while $\mathcal{X}_I^i \subset \mathcal{X}_I$ OR $\mathcal{X}_U^i \supset \mathcal{X}_U$ do
7      $i \leftarrow i + 1$
8      Choose $\mathcal{X}_I^i$, $\mathcal{X}_U^i$ such that $\mathcal{X}_I^i \supset \mathcal{X}_I^{i-1}$ or $\mathcal{X}_U^i \subset \mathcal{X}_U^{i-1}$
9      $\mathcal{X}_G^i \leftarrow \{x \in \overline{\mathcal{X}_U^{i-1}} \mid V_{i-1}(x) \le \beta_{i-1}\} \cup \mathcal{X}_G^0$
10     Choose $\alpha^i > \beta^i$ and $\epsilon^i > 0$
11     Train and verify controller $\pi_i$ and certificate $V_i$
         with witness $(\alpha^i, \beta^i, \epsilon^i)$ for the RWA task
         corresponding to $\mathcal{X}_I^i, \mathcal{X}_G^i$, and $\mathcal{X}_U^i$ using, e.g., the
         approach shown in Fig. 1
```

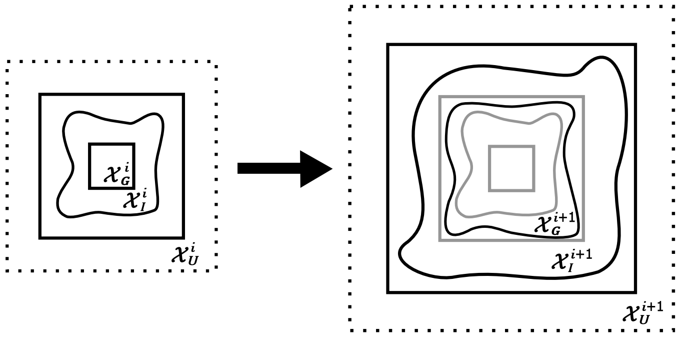


Fig. 2: Visualization of how consecutive certificates relate when building a CRWA certificate. Note that $\mathcal{X}_G$ need not be a subset of $\mathcal{X}_I$. The dotted lines indicate that the unsafe state region extends infinitely outside the solid line box. Wavy lines indicate outer boundaries for initial or goal regions.

sampled from $\mathcal{X}_I^i \setminus \mathcal{X}_G^i$. Otherwise, V_i might learn to assign values greater than β^i as much as possible in order to meet constraint (5), as opposed to appropriately assigning all states in $\mathcal{X}_I^i \setminus \mathcal{X}_G^i$ to have values less than β^i, due to an insufficient loss penalty for constraint (6). To ensure that states in the region $\mathcal{X}_I^i \setminus \mathcal{X}_G^i$ are included in the training data, we can identify states over constrained subspaces in $\mathcal{X}_I^i \setminus \mathcal{X}_G^i$, and then include in the data set those points as well as a random subset of their neighbors which likely lie in the same region.

Tradeoffs in choosing Intermediate Goals for CRWA certificates. It is possible to further reduce the state space for individual certificates in a CRWA certificate by using a more precise description of the goal states. In particular, we could set the goal states as follows:

$$\mathcal{X}_G^i = \{x \in \overline{\mathcal{X}_U^{i-1}} \mid V_{i-1}(x) \le \beta_{i-1}\} \cup \mathcal{X}_G^{i-1}. \quad (11)$$

However, using 11 leads to a linear increase in the number of DNNs that must be included during training and verification at each iteration of Alg. 1. This quickly becomes prohibitively expensive, especially for the verification step. We thus use the simpler formulation described above.

VI. Evaluation

A. Case Study

We evaluate our approach on the 2D docking task from [78],[7] in which a spacecraft is trained using DRL to navigate to a goal. More specifically, a DRL agent maneuvers a *deputy* spacecraft, controlled with thrusters that provide forces in the x and y directions. The deputy spacecraft attempts to safely navigate until it reaches a state that is in close proximity to a designated *chief* spacecraft, while obeying a distance-dependent safety constraint. We focus on this benchmark for several reasons: (i) it has been proposed and studied as a challenge problem in the literature [78], (ii) there exist natural safety and liveness properties for it; and (iii) existing approaches have been been unable to formally verify these properties.

System Dynamics. The system is modeled using the Clohessy-Wiltshire relative orbital motion linear approximation in the non-inertial Hill's reference frame, with the *chief* spacecraft lying at the origin [29], [50]. The state of the system, $\boldsymbol{x} = [x, y, \dot{x}, \dot{y}]^T$, includes the position in (x, y) and the velocities in each direction, $(\dot{x}, \dot{y})$. The control input is $\boldsymbol{u} = [F_x, F_y]$, where F_x and F_y are the thrust forces applied along the x and y directions, respetively. Each thrust force component is allowed to range between −1 and +1 Newtons (enforced with standard piecewise linear clipping). As in the original scenario [78], the spacecraft's mass, m, is 12kg. The continuous time state dynamics of the system are determined by the following ordinary differential equations (ODE), with $n = 0.001027$ rad/s:

$$\dot{\boldsymbol{x}} = [\dot{x}, \dot{y}, \ddot{x}, \ddot{y}]^T \quad (12)$$

$$\ddot{x} = 2n\dot{y} + 3n^2x + \frac{F_x}{m} \quad (13)$$

$$\ddot{y} = -2n\dot{x} + \frac{F_y}{m} \quad (14)$$

This, in turn, is converted to a discrete system (with a time-step of T) by numerically integrating the continuous time dynamics ODE:

$$\boldsymbol{x}(t_i + T) = \boldsymbol{x}(t_i) + \int_{t_i}^{t_i+T} \dot{\boldsymbol{x}}(\tau) d\tau \quad (15)$$

The discrete-time version has a closed-form solution that we use to generate successive states for the spacecraft.

Constraints and Terminal Conditions. To maintain safety, a distance-dependent constraint is imposed on the *deputy*

[7]Additional details on this case study are described in our recent related paper [70] and in the extended version of the current paper [69].

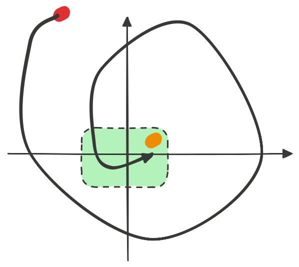


Fig. 3: A spiral trajectory: The DRL-controlled spacecraft (starting from the red point) eventually reaches the destination (orange) point within the docking region.

spacecraft's maximal velocity magnitude (while approaching the *chief*):

$$\sqrt{\dot{x}^2 + \dot{y}^2} \leq 0.2 + 2n\sqrt{x^2 + y^2} \qquad (16)$$

We construct a linear over-approximation of this safety constraints (with OVERT [83]). It can then be incorporated into the description of the unsafe region.

The goal is for the deputy to successfully dock with the chief without ever violating the velocity safety constraint. In the original benchmark [78], the docking (goal) region is defined as a circle of diameter $d = 1$m centered at the origin $(0, 0)$. In our evaluation, we use this same goal region for the initial training of the DRL controller. However, during the CEGIS iteration (i.e., the "Train" and "Verify" steps in Fig. 1), we use a conservative subset of this goal region, namely a square centered at the origin whose sides have length $l = 0.7$m. The reason for this is so that the goal region can easily be described using linear inequalities.

B. DNN architecture

We explored various architectures for the DNN used to control the (deputy) spacecraft. During this exploration phase, we trained each DNN architecture using the original Proximal Policy Optimization RL algorithm implemented by Ray RLlib as described in [78] (without any CEGIS iteration).

After training, we simulated each architecture on 4,000 random trajectories. Some selected results are shown in Table I. There are two main observations to take away from these results: (i) while a robust docking capability can be achieved fairly easily, even for small architectures, safety is more difficult and appears to not be robust, even for large architectures; (ii) in all cases, it takes an average of at least 50 steps to dock. The first observation suggests that verification of the liveness (docking) property should be feasible and that training a controller that verifiably achieves both safety and liveness is challenging. The second observation suggests that even state-of-the-art DNN verifiers are unlikely to be unable to fully verify the liveness property using the naive unrolling approach [9].

We also note that the spacecraft often exhibits highly nonlinear spiral trajectories (as depicted in Fig. 3), making DNN verification based on induction difficult, as it is difficult to find an inductive property over such irregular trajectories. These results help motivate the use of NLB certificates for formal verification of the desired properties. For the experiments below, we settled on a DNN architecture of two hidden layers with 20 neurons each, and a certificate architecture of two hidden layers with 30 neurons each. Both DNNs use ReLU activations for all hidden layers. The DNN sizes were chosen based on experimentation and the rough criterion that we wanted the smallest DNNs for which the CEGIS loop would converge in a reasonable amount of time.

TABLE I: Performance of various DNN architectures. Statistics are collected (per architecture) over $4,000$ trials, with a maximum trajectory length of $2,000$, initialized arbitrarily to set $x, y \in [-10, 10]$, but outside the docking region, and $\dot{x} = \dot{y} = 0$. The first column indicates the number of neurons per hidden layer.

DNN Architecture	Safety Success	Docking Success	Average Docking Steps
[4,4]	100	10	1,821
[8,8]	11	100	389
[16,16]	30	100	50
[32,32]	5	100	59
[64,64]	100	100	55
[64,64,64,64]	100	99	58
[200,200]	92	100	51

C. Implementation and Setup

The training and verification of the DRL controllers and certificates were carried out on a cluster of Intel Xeon E5-2637 machines, with eight cores of v4 CPUs, running the Ubuntu 20.04 operating system. Verification queries were dispatched using the *Marabou* DNN verifier [58], [87] (used in previous DNN safety research [10]–[14], [17], [23], [30], [61], [79]) as well as its Gurobi back end.

For training and verification of RWA, FRWA, and CRWA certificates, we use the following parameters: $\alpha = 1 + 10^{-5}$, $\beta = 1$, $\epsilon = 10^{-7}$ (the same for all certificates); $c_1 = -10$, $c_2 = 1.2$, $\delta_1 = 10^{-4} - 10^{-5}$, and $\delta_2 = 10^{-4} - 10^{-7}$. These values were determined to work well experimentally.

For weighting of the training objectives, we use $c_s = 1$ and $c_d = 10$. The rationale for this is that constraint (4) is much easier to satisfy than (5), so we use the weights to force the training to focus on (5).

In the CEGIS loop, a learning rate of 5×10^{-3} is used to train the first network iteration in the CEGIS loop, and for retraining, a learning rate of 10^{-4} is used, since we treat the incorporation of counterexamples as a "fine-tuning" step and do not want to overfit to the counterexamples. In the CEGIS loop, we train until a loss of 0 is achieved and then use the verification step to find counterexamples. We repeat this until there are no more counterexamples or a timeout (12 hours) is reached.

All of our experiments aim to solve RWA tasks, as defined in Definition 1. The system dynamics are those of the 2D spacecraft, as described in Section VI-A. RWA tasks are

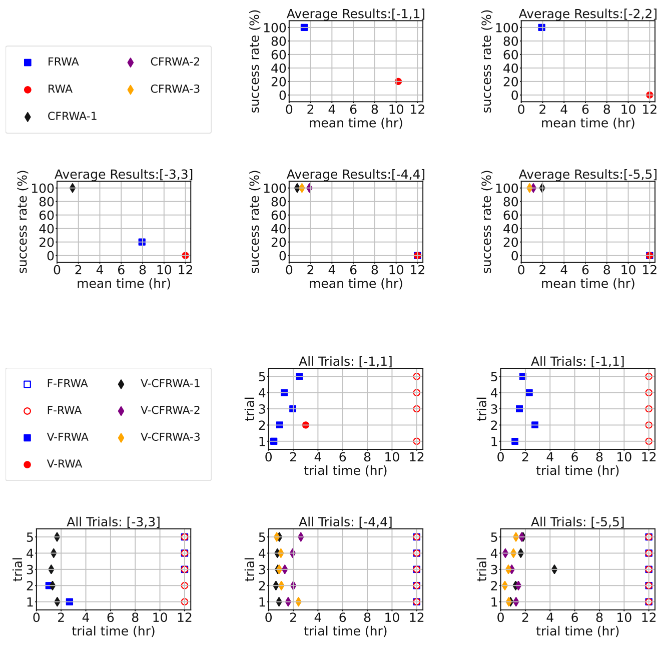


Fig. 4: The first 2 rows show average times and success rates for creating verified certificates over 5 trials. The bottom 2 rows show specific times for each trial, separated into failed ("F-") and verified ("V-") certificates. CFRWA-1, CFRWA-2, and CFRWA-3 refer to the first (up to) three CRWA tasks for the given starting region, corresponding, respectively to the first (up to) three rows for that starting region in Table II.

parameterized by $\mathcal{X}_G$, $\mathcal{X}_I$, and $\mathcal{X}_U$. For these sets of states, we typically use square regions centered at the origin. For convenience, we refer to the set $\{(x, y) \mid x, y \in [-a, a]\}$ with the abbreviation $[-a, a]$. For example, as outlined in Section VI-A, we set $\mathcal{X}_G = [-0.35, 0.35]$. We use different values for $\mathcal{X}_I$, depending on the experiment (indeed, this is the primary variable we vary in our experiments), but whenever $\mathcal{X}_I = [-a, a]$, we then set $\mathcal{X}_U = [-(a+1), a+1]$.

D. Experimental Results

RWA vs. FRWA. In our first set of experiments, we select a set of RWA tasks and train both RWA and FRWA certificates using our CEGIS loop.[8] We select five RWA tasks, where $\mathcal{X}_I$ is set to $[-i, i]$ for the ith task. For the RWA certificates, we follow the approach of [37], whereas our FRWA certificates are constructed as described in Sec. IV. In each case, we run five independent trials for each task.

The results are summarized in Fig. 4. The first two rows show, for each starting region, the number of successful runs (a run is successful if the CEGIS loop produces a fully verified controller/certificate pair within the 12 hour time limit) and the average time required for the successful runs. Results for RWA are shown as red circles and FRWA as blue squares (we explain the diamonds later). For example, for starting region $[-2, 2]$, all five trials are successful for FWRA, with an average time of 2 hours, whereas all five trials are unsuccessful for RWA. The bottom two rows show data from the same experiments, but here we show the time taken for each of the five trials. An unfilled circle or box represents a timeout.

The results suggest that FWRA has a clear advantage over standard RWA. In fact, RWA only succeeded once in producing any verified certificate, and only for the simplest starting region. On the other hand, our FRWA approach is able to produce certificates faster and for starting regions up to $[-3, 3]$. After that, both techniques time out.

Compositional Certificates. As demonstrated above, RWA and FRWA certificates quickly run into scalability challenges on our case study problem. For example, even with 5 tries and a 12 hour timeout, neither approach could produce a verified controller for the $[-4, 4]$ or $[-5, 5]$ starting regions.

Our second set of experiments demonstrates that this scalability challenge can be addressed with compositional certificates. We train a set of compositional certificates (each composed of multiple FRWA certificates) and report the results in Table II.

Each row of the table corresponds to a compositional certificate. The first column shows the value of $\mathcal{X}_I$ for this certificate. The next columns indicate the number n of composed certificates, the values of $\mathcal{X}_I^i$ for $0 \leq i < n-1$, and the cumulative time required for all but the last certificate. The next three columns give the minimum, mean, and maximum time required to produce the controller and certificate for the last stage of the compositional certificate (recall that we run five independent trials for all CEGIS loops). The next three columns show the minimum, mean, and maximum number of CEGIS iterations used, and the last column indicates how many of the trials succeeded. Note that when $n = 1$, the row corresponds to a single FRWA certificate.

The results clearly indicate that compositional certificates greatly improve scalability. Whereas the stand-alone certificates could not scale beyond $[-3, 3]$ in 12 hours, we were able to successfully produce a formally verified 5-stage certificate for $[-11, 11]$ in a little over 5.7 hours. It is also worth noting that we do get a significant benefit by running 5 independent CEGIS loops, as both the time and the number of loops can vary significantly from the minimum to the maximum. Nearly all of the CEGIS loops eventually completed—only the initial $[3, 3]$ region failed to complete all of its trials—suggesting that the compositional approach is also more stable and robust. This can also be seen in Fig. 4: for each starting region $[-a, a]$, the diamond point labeled CFRWA-i corresponds to the ith row containing $[-a, a]$ in column 1. We can see that, compared to the stand-alone RWA and FRWA certificates, the compositional certificates can be trained faster and with fewer failures.

[8]The Fossil 2.0 tool provides an implementation for computing the RWA certificates used in [37]. However, our definition of RWA is slightly different, and we use a different DNN verification tool, so we compare with our own implementation of RWA certificates to have a more meaningful comparison and to better isolate the contribution of the filtering technique.

TABLE II: Compositional certificate results. The columns indicate: the initial set for the final certificate, the size of the compositional certificate, the initial sets for all but the final certificate, the cumulative time for all but the final certificate, the total time (min, mean, and max) for training all certificates, and statistics for training the final certificate. We note that the cumulative time column is always equal to the corresponding value in the min column corresponding to the penultimate certificate. The wall time is the total time including the final controller/certificate. The CEGIS iterations and success stats are for the final controller/certificate only.

	Compositional Certificate			Wall Time (s)			CEGIS Iterations			
$\mathcal{X}_\mathbf{I}$	**n**	$\mathcal{X}_\mathbf{I}^\mathbf{0} \dots \mathcal{X}_\mathbf{I}^\mathbf{n-2}$	**Cumulative Time (s)**	**min(t)**	**mean(t)**	**max(t)**	**min(i)**	**mean(i)**	**max(i)**	**success (%)**
[-2,2]	1	N/A	0	4199	6890	8322	3	4.8	10	100
[-3,3]	1	N/A	0	3650	6644.5	9639	2	2.5	3	40
[-3,3]	2	[-2,2]	4199	8514	9421	10271	4	4.4	5	100
[-4,4]	2	[-3,3]	3650	5802	6374	6790	2	2.4	3	100
[-4,4]	2	[-2,2]	4199	8940	11026	13620	3	4.2	6	100
[-4,4]	3	[-2,2], [-3,3]	8514	10901	12829	17248	2	4.2	8	100
[-5,5]	2	[-3,3]	3650	6526	10716	19331	2	4.4	9	100
[-5,5]	3	[-3,3], [-4,4]	5802	7171	9884	11945	1	3.4	5	100
[-5,5]	4	[-2,2],[-3,3],[-4,4]	10901	12130	13710	15353	1	2.4	4	100
[-6,6]	3	[-2,2],[-4,4]	8940	13183	16384	20059	4	4.4	5	100
[-6,6]	4	[-3,3],[-4,4],[-5,5]	7171	9680	14027	32103	2	4.6	9	100
[-6,6]	5	[-2,2],[-3,3],[-4,4],[-5,5]	12130	18607	21768	24356	3	4.4	5	100
[-7,7]	3	[-3,3],[-5,5]	6526	9158	10171	10848	2	2.8	3	100
[-7,7]	5	[-3,3],[-4,4],[-5,5],[-6,6]	9680	11878	15967	23419	2	3.6	7	100
[-8,8]	4	[-2,2],[-4,4],[-6,6]	13183	16677	22623	33849	2	3.2	4	100
[-9,9]	4	[-3,3],[-5,5],[-7,7]	9158	12919	16013	18507	2	3.4	5	100
[-10,10]	5	[-2,2],[-4,4],[-6,6],[-8,8]	16677	18137	23421	30872	1	3.4	6	100
[-11,11]	5	[-3,3],[-5,5],[-7,7],[-9,9]	12919	20641	27860	32834	1	2.6	5	100

VII. Conclusion

In this work, we present a novel framework for formally verifying DRL-based controllers. Our approach leverages Neural Lyapunov Barrier certificates and demonstrates how they can be used to verify DNN-based controllers for complex systems. We use a CEGIS loop for training and formally verifying certificates, and we introduce filters for reach-while-avoid certificates, which simplify the training and verification process. We also introduce compositional certificates which use a sequence of simpler certificates to scale to large state spaces.

We demonstrate the merits of our approach on a 2D case study involving a DRL-controlled spacecraft which is required to dock in a predefined region, from any initialization point. We demonstrate that for small subdomains, our FRWA approach is strictly better than competing RWA-based certificate methods. Furthermore, we demonstrate that our compositional approach unlocks significant additional scalability.

In the future, we plan to extend our approach to be compatible with additional formal techniques (e.g., shielding against safety violations [6], [18], [31], [60], [74], [81], [89], and Scenario-Based Programming [32], [44], [47], [54], [57], [59], [92], [93]). We also plan to apply our approach to more challenging case studies with larger DRL controllers. We see this work as an important step towards the safe and reliable use of DRL in real-world systems.

VIII. Acknowledgements

This work was supported by AFOSR (FA9550-22-1-0227), the Stanford CURIS program, the NSF-BSF program (NSF: 1814369, BSF: 2017662), and the Stanford Center for AI Safety. The work of Amir was further supported by a scholarship from the Clore Israel Foundation. We thank Kerianne Hobbs (AFRL), Thomas Henzinger (ISTA), Chuchu Fan (MIT), and Songyuan Zhang (MIT) for useful conversations and advice which contributed to the success of this project.

References

[1] A. Abate, D. Ahmed, A. Edwards, M. Giacobbe, and A. Peruffo. Fossil: a software tool for the formal synthesis of lyapunov functions and barrier certificates using neural networks. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2021.

[2] A. Abate, D. Ahmed, M. Giacobbe, and A. Peruffo. Formal synthesis of lyapunov neural networks. *IEEE Control Systems Letters*, 5(3):773–778, 2020.

[3] A. Abate, S. Bogomolov, A. Edwards, K. Potomkin, S. Soudjani, and P. Zuliani. Safe reach set computation via neural barrier certificates. *arXiv preprint arXiv:2404.18813*, 2024.

[4] D. Ahmed, A. Peruffo, and A. Abate. Automated and sound synthesis of lyapunov functions with smt solvers. In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26*, pages 97–114. Springer, 2020.

[5] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 09 1987.

[6] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe Reinforcement Learning via Shielding. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence*, pages 2669–2678, 2018.

[7] A. Ames, X. Xu, J. W. Grizzle, and P. Tabuada. Control barrier function based quadratic programs for safety critical systems. *Trans. on Automatic Control*, 2017.

[8] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada. Control barrier functions: Theory and applications. In *European Control Conf.*, 2019.

[9] G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 607–627, 2023.

[10] G. Amir, Z. Freund, G. Katz, E. Mandelbaum, and I. Refaeli. veriFIRE: Verifying an Industrial, Learning-Based Wildfire Detection System. In *Proc. 25th Int. Symposium on Formal Methods (FM)*, pages 648–656, 2023.

[11] G. Amir, O. Maayan, T. Zelazny, G. Katz, and M. Schapira. Verifying Generalization in Deep Learning. In *Proc. 35th Int. Conf. on Computer Aided Verification (CAV)*, pages 438–455, 2023.

[12] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.

[13] G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 203–222, 2021.

[14] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 27–37, 2022.

[15] S. Bansal, M. Chen, S. Herbert, and C. J. Tomlin. Hamilton-Jacobi reachability: A brief overview and recent advances. In *Conf. on Decision and Control*, 2017.

[16] G. Basile and G. Marro. Controlled and conditioned invariant subspaces in linear system theory. *Journal of Optimization Theory and Applications*, 3:306–315, 1969.

[17] S. Bassan, G. Amir, D. Corsi, I. Refaeli, and G. Katz. Formally Explaining Neural Networks within Reactive Systems. In *Proc. 23rd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 10–22, 2023.

[18] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield Synthesis: - Runtime Enforcement for Reactive Systems. In *Proc. of the 21st Int. Conf. in Tools and Algorithms for the Construction and Analysis of Systems, (TACAS)*, volume 9035, pages 533–548, 2015.

[19] N. Boffi, S. Tu, N. Matni, J.-J. Slotine, and V. Sindhwani. Learning stability certificates from data. In *Conference on Robot Learning*, pages 1341–1350. PMLR, 2021.

[20] C. Brix, S. Bak, C. Liu, and T. T. Johnson. The fourth international verification of neural networks competition (vnn-comp 2023): Summary and results. *arXiv preprint arXiv:2312.16760*, 2023.

[21] L. Brunke, M. Greeff, A. W. Hall, Z. Yuan, S. Zhou, J. Panerati, and A. P. Schoellig. Safe learning in robotics: From learning-based control to safe reinforcement learning. *Annual Review of Control, Robotics, and Autonomous Systems*, 5:411–444, 2022.

[22] Y. Cao, H. Zhao, Y. Cheng, T. Shu, G. Liu, G. Liang, J. Zhao, and Y. Li. Survey on large language model-enhanced reinforcement learning: Concept, taxonomy, and methods, 2024.

[23] M. Casadio, E. Komendantskaya, M. Daggitt, W. Kokke, G. Katz, G. Amir, and I. Refaeli. Neural Network Robustness as a Verification Property: A Principled Case Study. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 219–231, 2022.

[24] F. Castañeda, J. J. Choi, B. Zhang, C. J. Tomlin, and K. Sreenath. Gaussian Process-based Min-norm Stabilizing Controller for Control-Affine Systems with Uncertain Input Effects. *arXiv*, Nov 2020.

[25] Y.-C. Chang and S. Gao. Stabilizing neural control using self-learned almost lyapunov critics. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1803–1809, 2021.

[26] Y.-C. Chang, N. Roohi, and S. Gao. Neural lyapunov control. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[27] J. Choi, F. Castañeda, C. J. Tomlin, and K. Sreenath. Reinforcement Learning for Safety-Critical Control under Model Uncertainty, using Control Lyapunov Functions and Control Barrier Functions. In *Robotics: Science and Systems*. Robotics: Science and Systems, Apr 2020.

[28] Y. Chow, O. Nachum, E. Duenez-Guzman, and M. Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8092–8101. Curran Associates, Inc., 2018.

[29] W. Clohessy and R. Wiltshire. Terminal guidance system for satellite rendezvous. *Journal of the aerospace sciences*, 27(9):653–658, 1960.

[30] D. Corsi, G. Amir, G. Katz, and A. Farinelli. Analyzing Adversarial Inputs in Deep Reinforcement Learning, 2024. Technical Report. https://arxiv.org/abs/2402.05284.

[31] D. Corsi, G. Amir, A. Rodriguez, C. Sanchez, G. Katz, and R. Fox. Verification-Guided Shielding for Deep Reinforcement Learning, 2024. Technical Report. http://arxiv.org/abs/2406.06507.

[32] D. Corsi, R. Yerushalmi, G. Amir, A. Farinelli, D. Harel, and G. Katz. Constrained Reinforcement Learning for Robotics via Scenario-Based Programming, 2022. Technical Report. https://arxiv.org/abs/2206.09603.

[33] C. Dawson, S. Gao, and C. Fan. Safe control with learned certificates: A survey of neural lyapunov, barrier, and contraction methods for robotics and control. *IEEE Transactions on Robotics*, 2023.

[34] C. Dawson, Z. Qin, S. Gao, and C. Fan. Safe nonlinear control using robust neural lyapunov-barrier functions. In *Conference on Robot Learning*, pages 1724–1735. PMLR, 2022.

[35] J. L. C. B. de Farias and W. M. Bessa. Intelligent control with artificial neural networks for automated insulin delivery systems. *Bioengineering*, 9(11):664, 2022.

[36] A. Edwards, A. Peruffo, and A. Abate. Fossil 2.0: Formal certificate synthesis for the verification and control of dynamical models. *arXiv preprint arXiv:2311.09793*, 2023.

[37] A. Edwards, A. Peruffo, and A. Abate. A general verification framework for dynamical and control models via certificate synthesis, 2023.

[38] J. F. Fisac, M. Chen, C. J. Tomlin, and S. S. Sastry. Reach-avoid problems with time-varying dynamics, targets and constraints. In *Proceedings of the 18th international conference on hybrid systems: computation and control*, pages 11–20, 2015.

[39] M. Ganai, Z. Gong, C. Yu, S. L. Herbert, and S. Gao. Iterative reachability estimation for safe reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

[40] M. Ganai, C. Hirayama, Y.-C. Chang, and S. Gao. Learning stabilization control from observations by learning lyapunov-like proxy models. *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2913–2920, 2023.

[41] S. Gao, S. Kong, and E. M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In *International conference on automated deduction*, pages 208–214. Springer, 2013.

[42] P. Giesl and S. Hafstein. Review on computational methods for lyapunov functions. *Discrete and Continuous Dynamical Systems-B*, 20(8):2291–2331, 2015.

[43] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[44] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th ACM Annual Conf. on Innovation and Technology in Computer Science Education (ITCSE)*, pages 198–203, 2012.

[45] D. Grande, E. Anderlini, A. Peruffo, and G. Salavasidis. Augmented neural lyapunov control. *IEEE Access*, 2023.

[46] D. Grande, D. Fenucci, A. Peruffo, E. Anderlini, A. B. Phillips, G. Thomas, and G. Salavasidis. Systematic synthesis of passive fault-tolerant augmented neural lyapunov control laws for nonlinear systems. In *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 5851–5856. IEEE, 2023.

[47] J. Greenyer, D. Gritzner, G. Katz, and A. Marron. Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *Proc. 19th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 16–23, 2016.

[48] W. Haddad and V. Chellaboina. Nonlinear dynamical systems and control: A lyapunov-based approach. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*, 01 2008.

[49] T. Hester, M. Quinlan, and P. Stone. A real-time model-based reinforcement learning architecture for robot control, 2011.

[50] G. W. Hill. Researches in the lunar theory. *American journal of Mathematics*, 1(1):5–26, 1878.

[51] K.-C. Hsu, V. Rubies-Royo, C. J. Tomlin, and J. F. Fisac. Safety and liveness guarantees through reach-avoid reinforcement learning. In *Proceedings of Robotics: Science and Systems*, Virtual, 7 2021.

[52] T. Huang, S. Gao, and L. Xie. A neural lyapunov approach to transient stability assessment of power electronics-interfaced networked microgrids. *IEEE transactions on smart grid*, 13(1):106–118, 2021.

[53] Z. Jarvis-Wloszek, R. Feeley, Weehong Tan, Kunpeng Sun, and A. Packard. Some controls applications of sum of squares programming. In *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, volume 5, pages 4676–4681 Vol.5, Dec 2003.

[54] G. Katz. Guarded Deep Learning using Scenario-Based Modeling. In *Proc. 8th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 126–136, 2020.

[55] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.

[56] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.

[57] G. Katz and A. Elyasaf. Towards Combining Deep Learning, Verification, and Scenario-Based Programming. In *Proc. 1st Workshop on Verification of Autonomous and Robotic Systems (VARS)*, pages 1–3, 2021.

[58] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.

[59] G. Katz, A. Marron, A. Sadon, and G. Weiss. On-the-Fly Construction of Composite Events in Scenario-Based Modeling Using Constraint Solvers. In *Proc. 7th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 143–156, 2019.

[60] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.

[61] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.

[62] M. Landers and A. Doryab. Deep reinforcement learning verification: A survey. *ACM Comput. Surv.*, 55(14s), jul 2023.

[63] B. Li, S. Wen, Z. Yan, G. Wen, and T. Huang. A survey on the control lyapunov function and control barrier function for nonlinear-affine control systems. *IEEE/CAA Journal of Automatica Sinica*, 10(3):584–602, 2023.

[64] Y. Li. Deep Reinforcement Learning: An Overview, 2017. Technical Report. http://arxiv.org/abs/1701.07274.

[65] J. Lu. Protein folding structure prediction using reinforcement learning with application to both 2d and 3d environments. In *Proceedings of the 5th International Conference on Computer Science and Software Engineering*, CSSE '22, page 534–542, New York, NY, USA, 2022. Association for Computing Machinery.

[66] A. M. Lyapunov. The general problem of motion stability. *Annals of Mathematics Studies*, 17(1892), 1892.

[67] Z. Lyu, C. Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044, 2020.

[68] A. Majumdar and R. Tedrake. Funnel libraries for real-time robust feedback motion planning. *The International Journal of Robotics Research*, 36(8):947–982, 2017.

[69] U. Mandal, G. Amir, H. Wu, I. Daukantas, F. Newell, U. Ravaioli, B. Meng, M. Durling, M. Ganai, T. Shim, G. Katz, and C. Barrett. Formally Verifying Deep Reinforcement Learning Controllers with Lyapunov Barrier Certificates, 2024. Technical Report. https://arxiv.org/abs/2405.14058.

[70] U. Mandal, G. Amir, H. Wu, I. Daukantas, F. Newell, U. Ravaioli, B. Meng, M. Durling, K. Hobbs, M. Ganai, T. Shim, G. Katz, and C. Barrett. Safe and Reliable Training of Learning-Based Aerospace Controllers, 2024. Technical Report. http://arxiv.org/abs/2407.07088.

[71] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[72] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[73] A. Peruffo, D. Ahmed, and A. Abate. Automated and formal synthesis of neural barrier certificates for dynamical models. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 370–388. Springer, 2021.

[74] S. Pranger, B. Könighofer, M. Tappler, M. Deixelberger, N. Jansen, and R. Bloem. Adaptive Shielding under Uncertainty. In *American Control Conference, (ACC)*, pages 3467–3474, 2021.

[75] Z. Qin, T.-W. Weng, and S. Gao. Quantifying safety of learning-based self-driving control using almost-barrier functions. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12903–12910. IEEE, 2022.

[76] Z. Qin, K. Zhang, Y. Chen, J. Chen, and C. Fan. Learning safe multi-agent control with decentralized neural barrier certificates. In *ICLR*, 2021.

[77] M. Rada and M. Cerny. A new algorithm for enumeration of cells of hyperplane arrangements and a comparison with avis and fukuda's reverse search. *SIAM Journal on Discrete Mathematics*, 32(1):455–473, 2018.

[78] U. J. Ravaioli, J. Cunningham, J. McCarroll, V. Gangal, K. Dunlap, and K. L. Hobbs. Safe reinforcement learning benchmark environments for aerospace control systems. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–20. IEEE, 2022.

[79] I. Refaeli and G. Katz. Minimal Multi-Layer Modifications of Deep Neural Networks, 2021. Technical Report. https://arxiv.org/abs/2110.09929.

[80] S. M. Richards, F. Berkenkamp, and A. Krause. The lyapunov neural network: Adaptive stability certification for safe learning of dynamical systems. In *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 466–476, 29–31 Oct 2018.

[81] A. Rodriguez, G. Amir, D. Corsi, C. Sanchez, and G. Katz. Shield Synthesis for LTL Modulo Theories, 2024. Technical Report. http://arxiv.org/abs/2406.04184.

[82] P. Samanipour and H. A. Poonawala. Stability analysis and controller synthesis using single-hidden-layer relu neural networks. *IEEE Transactions on Automatic Control*, 2023.

[83] C. Sidrane, A. Maleki, A. Irfan, and M. J. Kochenderfer. Overt: An algorithm for safety verification of neural network control policies for nonlinear systems. *Journal of Machine Learning Research*, 23(117):1–45, 2022.

[84] O. So and C. Fan. Solving stabilize-avoid optimal control via epigraph form and deep reinforcement learning. In *Proceedings of Robotics: Science and Systems*, 2023.

[85] V. Talpaert, I. Sobh, B. R. Kiran, P. Mannion, S. Yogamani, A. El-Sallab, and P. Perez. Exploring applications of deep reinforcement learning for real-world autonomous driving systems, 2019.

[86] M. Tong, C. Dawson, and C. Fan. Enforcing safety for vision-based controllers via control barrier functions and neural radiance fields. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10511–10517. IEEE, 2023.

[87] H. Wu, O. Isac, A. Zeljić, T. Tagomori, M. Daggitt, W. Kokke, I. Refaeli, G. Amir, K. Julian, S. Bassan, et al. Marabou 2.0: A versatile formal analyzer of neural networks. *arXiv preprint arXiv:2401.14461*, 2024.

[88] J. Wu, A. Clark, Y. Kantaros, and Y. Vorobeychik. Neural lyapunov control for discrete-time systems. *Advances in Neural Information Processing Systems*, 36:2939–2955, 2023.

[89] M. Wu, J. Wang, J. Deshmukh, and C. Wang. Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems. In *Proc. 19th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 129–137, 2019.

[90] X. Xu, P. Tabuada, J. W. Grizzle, and A. D. Ames. Robustness of control barrier functions for safety critical control. *Int. Federation of Automatic Control*, 2015.

[91] Y. Yang, Y. Jiang, Y. Liu, J. Chen, and S. E. Li. Model-free safe reinforcement learning through neural barrier certificate. *IEEE Robotics and Automation Letters*, 2023.

[92] R. Yerushalmi, G. Amir, A. Elyasaf, D. Harel, G. Katz, and A. Marron. Scenario-Assisted Deep Reinforcement Learning. In *Proc. 10th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 310–319, 2022.

[93] R. Yerushalmi, G. Amir, A. Elyasaf, D. Harel, G. Katz, and A. Marron. Enhancing Deep Reinforcement Learning with Scenario-Based Modeling. *SN Computer Science*, 4(2):156, 2023.

[94] D. Yu, H. Ma, S. Li, and J. Chen. Reachability constrained reinforcement learning. In *International Conference on Machine Learning*, pages 25636–25655. PMLR, 2022.

[95] H. Yu, C. Hirayama, C. Yu, S. Herbert, and S. Gao. Sequential neural barriers for scalable dynamic obstacle avoidance. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11241–11248. IEEE, 2023.

[96] H. Zhang, J. Wu, Y. Vorobeychik, and A. Clark. Exact verification of relu neural control barrier functions. *Advances in Neural Information Processing Systems*, 36, 2024.

[97] R. Zhou, T. Quartz, H. De Sterck, and J. Liu. Neural lyapunov control of unknown nonlinear systems with stability guarantees. *Advances in Neural Information Processing Systems*, 35:29113–29125, 2022.

Formal Methods in Computer-Aided Design 2024

Leveraging LLMs for Program Verification

Adharsh Kamath*, Nausheen Mohammed*, Aditya Senthilnathan†, Saikat Chakraborty‡, Pantazis Deligiannis*, Shuvendu K. Lahiri‡, Akash Lal*, Aseem Rastogi*, Subhajit Roy§, Rahul Sharma*
* Microsoft Research, Bangalore, India
† Cornell University, Ithaca, USA
‡ Microsoft Research, Redmond, USA
§ Indian Institute of Technology, Kanpur, India

***Abstract*—We investigate code reasoning skills of Large Language Models (LLMs) in the context of formal program verification. Specifically, we look at the problem of inferring loop invariants as well as ranking functions for proving safety properties and loop termination, respectively. We demonstrate how emergent capabilities of LLMs can be exploited through a combination of prompting techniques as well as by using them in conjunction with symbolic algorithms. We curate and contribute a dataset of verification problems inspired by past work. We perform a rigorous evaluation on this dataset to establish that LLMs have the potential of improving state-of-the-art in program verification.**

I. Introduction

Formal verification seeks to establish a proof of correctness of a program with respect to a given property. Broadly speaking, this involves *proof construction* (e.g., finding loop invariants), and *proof checking* (e.g., establishing their inductiveness). While proof checking has benefited from mechanical automation enabled by SMT solvers, proof construction still requires *ingenuity* and has been harder to automate.

The *guess-and-check* methodology seeks to reduce burden on the proof construction by allowing it to *guess* a proof that potentially may have mistakes. The soundness comes solely from proof checking. This methodology has allowed for Machine-learning (ML) based techniques to enter program verification. While it is hard for ML techniques to guarantee soundness, it can still be a source of good "guesses" on why a given program is correct, and better guesses lead to faster verification. Work in this space includes generating data from program executions and guess invariants through classical learning techniques [1], [2], active learning over decision trees [3], continuous logic networks [4], [5], as well as training neural networks to directly predict invariants from program text [6], [7]. The trend of training models for individual tasks, thus requiring independent datasets, is changing with the advent of Large Language Models (LLMs) and this forms the inspiration for our paper.

Latest foundational models such as GPT-4 [8], PaLM-2 [9], Llama-2 [10] have been trained on vasts amount of data, and have shown remarkable ability in solving a diverse set of tasks. One can supply a set of instructions in natural language to guide the model towards a certain task of interest [11]. LLMs are already aiding many software developers in writing code [12], [13]. We study the use of these foundational models for constructing proofs that can be discharged by a formal proof checker, following the guess-and-check methodology.

We study two different verification tasks, one on safety verification and another on proving program termination. Safety verification requires finding inductive invariants for loops as well as pre-post conditions for procedures, so that the given assertions in a program can be proved safe. In termination, the goal is to find a ranking function, as well as supporting invariants, to prove termination of loops. We curate a dataset of programs in the C language for these tasks and build an LLM-based toolchain, called Loopy, for proof generation. The proofs are discharged by an off-the-shelf formal checker; in our implementation, we use Frama-C [14] because it directly supports C programs with invariant annotations.

Loopy is based on two key aspects that help make effective use of LLM capabilities. The first is *prompt engineering* that encodes a set of instructions to describe the different tasks to an LLM. For instance, the prompt for program termination includes a definition of ranking functions in natural language. Ranking functions come in various forms, such as *lexicographic ranking functions* and *multi-phase ranking functions*; we demonstrate that LLMs are capable of producing such ranking functions when prompted with their definitions. We also introduce the concept of *nudging* where additional generic instructions are added in natural language to help the LLM generate the required proof artifact. For instance, the LLM is "encouraged" to use implications for dealing with conditional code, inferring bounds of loop variables, or producing invariants on unmodified parts of an array.

The second key aspect is the interplay between an LLM and a symbolic (formal) tool. We find that LLMs are better at generating ingredients of an inductive invariant than they are at generating the whole invariant. Consequently, Loopy uses the Houdini algorithm [15] to weed out incorrect guesses and converge to a correct inductive subset from the set of LLM-generated guesses. Houdini only uses a linear number of calls to the checker (linear in the number of candidate invariants).

Contributions: We have curated a dataset of C programs for multiple different program verification tasks.[1] We also build and evaluate a tool called Loopy for effectively leveraging LLM capabilities on these tasks. We present an evaluation

[1] https://github.com/microsoft/loop-invariant-gen-experiments

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_16

with multiple LLMS: GPT-4 [8], GPT-3.5 [16], and Code Llama [10], and compare the performance of LOOPY against a state-of-the-art symbolic baseline. Our results establish that LLMs have the potential of improving state-of-the-art in program verification. LOOPY is able to out-perform existing symbolic tools on several benchmarks.

II. VERIFICATION TASKS AND DATASETS

We define a verification task as a C program along with a property of interest that must be established for the program. Our choice of the C programming language is based on the availability of the benchmarks as well as a formal checker (Frama-C [14]). Frama-C defines a language called ACSL [17] for writing annotations (assertions, invariants, ranking functions, etc.) as comments in C programs. We consider two kinds of verification tasks: Safety verification and Termination checking.

a) Safety verification: In this category of benchmarks, each of the C programs have embedded assertions. The goal is to come up with ACSL annotations that help Frama-C prove those assertions. We obtained these benchmarks from multiple sources, including Code2Inv [7], Accelerating Invariant Generation [18], Data-Driven CHC solver [19], Fluid Updates [20], Diffy [21], as well as the SV-COMP repository [22].

We perform basic filtering on these benchmarks. We discard programs that are known to be incorrect (i.e., the assertion does not hold). We also remove programs that are greater than 500 lines of code. This allows us to fit the entire program inside a single LLM query, helping us to focus on the code reasoning capabilities of LLMs. We then create three exclusive datasets, based on certain program features, as described below.

The first dataset consists of 469 programs that use only scalar types (either signed or unsigned) with a single `main` procedure with one loop. This category of benchmarks exercises basic mathematical reasoning, without bringing in concerns of modeling pointers, heap semantics, or quantified invariants. The second dataset consists of 31 programs with recursion, only scalar types, and no loops. These benchmarks have minimum 1, maximum 4, and average 1.4 non-`main` methods. The third dataset consists of 169 programs with a single method and at least one array or pointer. We are restricted by limitations of Frama-C to deal with such programs because it currently lacks support for dynamic memory allocation. We manually remove memory allocation from programs where it is not important. These programs have minimum 1, maximum 13, and an average of 4.4 loops per program.

b) Termination checking: The goal here is to infer a ranking function (also called a loop variant) for a loop that proves its termination. A ranking function is an integer-valued expression defined over program variables that is bounded below by 0 and strictly decreases in each iteration. We collect benchmarks from The Termination Competition [23] and from Shi et. al. [24]. We filter these programs, retaining programs that each consist of only scalar variables, single method, no assertions, and a single loop (that we believe is terminating). This set consists of 281 benchmarks.

We remove any comments in all the programs in our datasets because they could potentially provide hints to the LLMs.

c) Summary of the results: We show examples of programs in each of our datasets in Figure 1. All the comments in these examples are LOOPY-generated output (massaged slightly for conciseness), which in each case suffices to complete the corresponding verification task. Figure 2 summarizes LOOPY's performance across these datasets. The column Total is the total number of benchmarks in the dataset. The column "Vanilla LLMs" refers to the number of benchmarks that GPT-4 is able to solve with a basic prompt (and multiple completions, we detail these concepts in later sections). These numbers show the raw LLM performance on the corresponding tasks. Column LOOPY is the number of benchmarks solved by our LLM-based toolchain; a significant increase over raw LLM performance. The next two columns provide a comparison against a symbolic baseline. We show the performance of Ultimate Automizer [25], one of the tools that routinely wins medals in SV-COMP. The last column is the number of benchmarks that could be solved by either LOOPY or Ultimate, showcasing the potential of LLMs in improving the state-of-the-art.

III. INDUCTIVE LOOP INVARIANT INFERENCE

This section considers the problem of inferring inductive loop invariants. Guided by empirical observations, we propose three techniques that can be used to augment LLMs for such tasks: (a) providing domain-specific instructions to the LLMs, (b) filtering incorrect LLM outputs with an adaptation of the Houdini algorithm, and (c) using LLMs to repair the incorrect invariants. We find that these three techniques significantly improve the ability of LLMs to infer loop invariants. While we use GPT-4 for most of our experiments, we also present a comparison with GPT-3.5-Turbo and CodeLlama-34b-Instruct [10] (an open source LLM) in III-G.

A. The problem

Consider a C-like imperative language. Let S denote statements written in this language. Hoare triples $\{P\} \;\; S \;\; \{Q\}$, where P and Q are logical propositions over program variables in some underlying logic, are assertions interpreted as: if P holds before executing the statement S, then Q holds after its completion [26] (S may not terminate though). Inductive loop invariants, I, are logical summaries for loop statements in the language, used to prove the corresponding Hoare triples. For example, for $while$, following is an inference rule to derive a Hoare triple (B denotes boolean expressions):

$$\frac{P \Rightarrow I \qquad \{I \wedge B\} \;\; S \;\; \{I\} \qquad I \wedge \neg B \Rightarrow Q}{\{P\} \;\; while\ B\ do\ S \;\; \{Q\}}$$

Inductive loop invariant I is a logical proposition that (a) holds at the beginning of the loop ($P \Rightarrow I$), (b) is preserved by the loop body ($\{I \wedge B\} \;\; S \;\; \{I\}$), and (c) implies the postcondition of the loop ($I \wedge \neg B \Rightarrow Q$). We assume that the qualifier "inductive" is implicit when we talk about invariants in this paper, and hence drop it for convenience.

```
void main() {
  int k = 100, b = 0;
  int i = 0, j = 0, n;
  // @invariant 0 <= n <= 2k
  // @invariant n % 2 == b
  // @invariant b == 0 ==> i == j
  // @invariant b == 1 ==> i == j - 1
  for (n = 0; n < 2*k; n++) {
    if (b) i++; else j++;
    b = !b;
  }
  assert (i == j);
}

(a) Loop invariant
```

```
void main() {
  int x = unknown(), y = unknown(), z = unknown();
  // multi-phase ranking function
  // [z; y; x]
  while (x >= 0) {
    if (unknown()) x = x + y;
    else x = x + z;
    y = y + z; z = z - 1;
  }
}

(c) ranking function for loop termination
```

```
void main()
{
  int N = unknown ();
  if (N <= 0) return;
  int i, a[N];
  // @invariant 0 <= i <= N;
  // @invariant \forall k. 0 <= k < i ==> a[k] == 3;
  // @assigns i, a[0..N-1];
  for (i = 0; i < N; i++) { a[i] = 3; }

  // @invariant 0 <= i <= N;
  // @invariant \forall j. i <= j < N ==>
  //                       a[j] == \at(a[j], Entry);
  // @invariant \forall j. 0 <= j < i ==>
  //                       a[j] == 1 || a[j] == 2;
  // @assigns i, a[0..N-1];
  for (i = 0; i < N; i++) {
    if (N % (i + 1) == 0) a[i] = a[i] - 1;
    else a[i] = a[i] - 2;
  }
  for (i = 0; i < N; i++) { assert (a[i] <= 2); }
}

(b) Loop invariant over arrays
```

Fig. 1: Example programs along with LOOPY-generated annotations. Each program verifies with Frama-C.

Benchmark	Features	Total	Vanilla LLMs	LOOPY	Ultimate	LOOPY + Ultimate
Scalar loops	(1, 1)	469	237 (51%)	398 (85%)	430 (92%)	461 (98%)
Array loops	(1, $\geq$1)	169	60 (36%)	127 (75%)	12 (7%)	128 (75%)
Recursion	($\geq$1, 0)	31	14 (45%)	16 (52%)	20 (65%)	23 (74%)
Termination	(1, 1)	281	49 (17%)	181 (64%)	236 (84%)	255 (91%)

Fig. 2: Summary of LOOPY results. Features are (#methods, #loops).

Automatically synthesizing loop invariants is one of the classical problems in program verification. Our goal is to use and evaluate LLMs for this task. Interactions with LLMs happen via *prompts*. Prompts are textual instructions for LLMs to perform a task. LLMs respond to prompts with textual answers. LLMs may also be instructed to generate multiple responses (commonly called as *completions*) for one prompt. For our tasks, we design *prompt templates* that contain common instructions for LLMs to infer loop invariants, and template holes for the exact program. For each benchmark, we instantiate the template hole with the benchmark program.

B. Basic algorithm

Figure 3 shows our basic algorithm for loop invariant inference using LLMs; in the subsequent sections, we will refine it with additional techniques. The algorithm takes as input a program $\mathcal{P}$, a prompt template $\mathcal{M}$, and the number of completions $\mathcal{N}_c$. It either returns Success $\mathcal{I}$, where $\mathcal{I}$ is a set of propositions such that $\bigwedge_{i \in \mathcal{I}} i$ is a loop invariant strong enough to prove the assertions in $\mathcal{P}$, or it returns Failure when it cannot infer a sufficiently strong loop invariant.

```
procedure INFERENCE($\mathcal{P}, \mathcal{M}, \mathcal{N}_c$)
    while $0 < \mathcal{N}_c$ do
        $\mathcal{I} \leftarrow \mathcal{L}(\mathcal{M}[\mathcal{P}])$
        $b, \_, \_ \leftarrow \mathcal{O}(\mathcal{P}, \mathcal{I})$
        if $b$ then return Success $\mathcal{I}$
        else $\mathcal{N}_c \leftarrow \mathcal{N}_c - 1$
    return Failure
```

Fig. 3: Algorithm for invariant inference using LLMs

To check the output of LLMs, the algorithm relies on an Oracle $\mathcal{O}$. The oracle takes as input the program $\mathcal{P}$ and the set $\mathcal{I}$. It returns as output a triple $(b, \mathcal{I}_S, \mathcal{I}_{NI})$, where:

1) b is a boolean value, true if $\mathcal{P}$ verifies with the loop invariant $\bigwedge_{i \in \mathcal{I}} i$, false otherwise.
2) $\mathcal{I}_S \subseteq \mathcal{I}$ is the set of invariants that exhibit parsing errors; when b is true, $\mathcal{I}_S$ is empty.
3) $\mathcal{I}_{NI} \subseteq \mathcal{I}$ is the set of invariants for which the oracle cannot establish the inductiveness property. This can happen for two reasons: (a) when the invariant does not

hold at the beginning of the loop, or (b) when the loop body does not maintain the invariant. When b is true or when $\mathcal{I}_S$ is non-empty, $\mathcal{I}_{NI}$ is empty.

We assume that the oracle is sound, i.e., if it returns true, the assertions in $\mathcal{P}$ can be proven in the underlying logic using the loop invariant $\bigwedge_{i\in\mathcal{I}} i$. The basic algorithm does not use $\mathcal{I}_S$ and $\mathcal{I}_{NI}$.

Given these notations, the algorithm is a straightforward loop that prompts the LLM with the prompt template instantiated with the program ($\mathcal{M}[\mathcal{P}]$) until it either finds a loop invariant or runs out of the number of completions to try. Soundness of the algorithm follows from the soundness of the oracle $\mathcal{O}$.

For our experiments, we instantiate $\mathcal{O}$ with Frama-C, configured to use only the WP plugin for verifying ACSL annotations. Under these settings, Frama-C does not attempt to infer the invariants by itself; it is focused on verifying the correctness of supplied loop invariants as well as the assertions in the input program. Additionally, we configure the WP plugin to use Z3 [27], Alt-Ergo [28], and CVC4 [29] as the external provers, with a timeout of 3 seconds.

C. Basic prompt

We evaluate the algorithm with a basic prompt template, $\mathcal{M}_0$, shown below. Here, the {{ code }} section is the template hole for the program. Notably, the template does not explain to LLM what loop invariants are or provide any detailed instructions for inferring them. It does, however, provide instructions to format the output in the ACSL syntax; this helps in automating the checking process.

```
Consider the following C program:
{{ code }}
Output the loop invariants for the loop in the program above. Output
all the loop invariants in one code block. E.g.,
/*@
    loop invariant i1;
    loop invariant i2;
*/
```

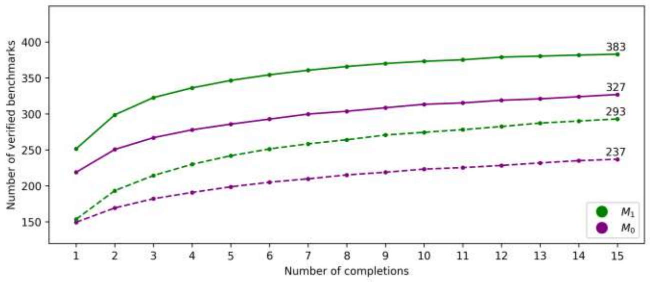


Fig. 4: Performance of LOOPY with and without Houdini (solid and dashed lines, resp.) for the two prompt templates

Figure 4 (dashed line for prompt $\mathcal{M}_0$) shows experimental results with GPT-4. It plots the success rate (number of verified benchmarks) as the number of completions is varied from 1 to 15. We account for the stochastic nature of LLMs in the standard way using the pass@k metric [30]: we first generate the maximum number of completions (15) and compute its success rate. Then, the success rate for $k < 15$ completions is obtained as the expectation over a random sample of size k out of the 15 completions. As the figure shows, with 15 completions, GPT-4 is able to solve ∼50% (237/469) of the benchmarks. Multiple completions help, though there are diminishing returns after ∼8 completions. The experiment shows that even without any specialized instructions or techniques, GPT-4 is able to solve a non-trivial fraction of the benchmarks.

D. Prompt with domain-specific instructions

On manual inspection of the failure cases with $\mathcal{M}_0$, we observe that the model makes several low-level mistakes, such as using variables or functions that are not defined, using conditional statements in the invariants, etc. We also notice that the model misses certain common invariant expressions, such as bounding a variable with its minimum and maximum values or relations between variables themselves. Consider the loop invariant example in Figure 1(a). With $\mathcal{M}_0$, the LLM only outputs `0 ≤ n≤ 2*k` and `n % 2 == b`. While both these are valid invariants, they are not sufficient to prove the assertion, as they don't capture the relationship between `i` and `j`, which is conditional on the value of `b`.

To account for such failures, we design a prompt template $\mathcal{M}_1$ that provides more detailed instructions to the LLM. Specifically, it explains to the LLM, in natural language, what a loop invariant is, and provides some heuristics about how to come up with a loop invariant. The full prompt $\mathcal{M}_1$ is given below.

```
You are a helpful AI software assistant that reasons about how
code behaves. Given a program, you can find loop invariants,
which can then be used to verify some property in the program.
Frama-C is a software verification tool for C programs. The input to
Frama-C is a C program file with ACSL (ANSI/ISO C Specification
Language) annotations. For the given program, find the necessary
loop invariants of the while loop to help Frama-C verify the post-
condition.
Instructions:
  • Make a note of the pre-conditions or variable assignments in
    the program.
  • Analyze the loop body and make a note of the loop condition.
  • Output loop invariants that are true
      (i)  before the loop execution,
     (ii)  in every iteration of the loop and
    (iii)  after the loop termination,
    such that the loop invariants imply the post condition.
  • If a loop invariant is a conjunction, split it into its parts.
  • Output all the loop invariants in one code block.
    For example:
    ```
    /*@
    loop invariant i1;
    loop invariant i2;
    */
    ```
Rules: **Do not use variables or functions that are not declared
in the program.** **Do not make any assumptions about
functions whose definitions are not given.** **All undefined
variables contain garbage values. Do not use variables that
have garbage values.** **Do not use keywords that are not
supported in ACSL annotations for loops.** **Variables that are
not explicitly initialized, could have garbage values. Do not make
any assumptions about such values.** **Do not use the \at(x,
```

Pre) notation for any variable x.** **Do not use non-deterministic function calls.**

Consider the following C program:
```
{{ code }}
```
You are allowed to use implication to take care of the conditional nature of the code. Use implication (==>) instead of using if-then. For all variables, add conjunctions that bound the maximum and minimum values that they can take, if such bounds exist. If a variable is always equal to or smaller or larger than another variable, add a conjunction for their relation. If the assertion is guarded by a condition, use the guard condition in an implication. If certain variables are non-deterministic at the beginning or end of the loop, use an implication to make the invariant trivially true at that location. Output the loop invariants for the loop in the program above. Let's think step by step.

To evaluate $\mathcal{M}_1$, we repeat the same experiment as before with $\mathcal{M}_1$ and GPT-4; see dashed line for $\mathcal{M}_1$ in Figure 4. GPT-4 is able to solve 293 benchmarks, 23% more than $\mathcal{M}_0$, demonstrating the effectiveness of detailed prompt instructions. For the example in Figure 1(a), with $\mathcal{M}_1$, LLM outputs the final two invariants $\mathsf{b} == 0 \Rightarrow \mathsf{i} == \mathsf{j}$ and $\mathsf{b} == 1 \Rightarrow \mathsf{i} == \mathsf{j} - 1$, which are sufficient to verify the example.

E. Pruning incorrect invariants with Houdini

In many cases, we observe that the LLM output contains the required invariants but they are mixed with other output expressions that are either syntactically invalid or are not valid loop invariants. Further, the required invariants may be spread across multiple completions. In both these cases, the basic algorithm fails.

The program below is one such example. A candidate loop invariant for this is $0 \leq \mathsf{x} < \mathsf{n}$. We observe that in most completions, LLM outputs $0 \leq \mathsf{x}$ and $\mathsf{x} \leq \mathsf{n}$ as invariants, while there are some completions in which it outputs $\mathsf{input} == 0 \Rightarrow \mathsf{x} < \mathsf{n}$ and in some other it outputs $\mathsf{input} \neq 0 \Rightarrow \mathsf{x} < \mathsf{n}$. All the completions together have the correct components, $0 \leq \mathsf{x}$, $\mathsf{input} == 0 \Rightarrow \mathsf{x} < \mathsf{n}$, and $\mathsf{input} \neq 0 \Rightarrow \mathsf{x} < \mathsf{n}$, but no single completion is correct in itself.

```
int n = unknown(); if (n <= 0) return;
int x = 0, input = unknown();
while (1) {
  if (input) { x = x + 1; if (x >= n) break; }
  input = unknown();
}
assert (x == n);
```

To handle such cases, we augment our basic algorithm with Houdini [15] to efficiently prune the incorrect outputs; Figure 5 shows the new algorithm. The algorithm maintains a set $\mathcal{I}_u$ of all the invariants output by the LLM across all the completions. If none of the completions succeed, the algorithm invokes the Houdini procedure with $\mathcal{I}_u$, and returns the result of the Houdini procedure.

The Houdini procedure tries to find a subset of $\mathcal{I}_u$ that is inductive and is sufficient to verify $\mathcal{P}$. While the number of possible subsets of $\mathcal{I}_u$ is exponential in the size of $\mathcal{I}_u$, it turns out that one can do this check with only a linear number of calls to the oracle (linear in the size of $\mathcal{I}_u$) [15]. Figure 6 shows

```
1:  procedure INFERENCE($\mathcal{P}, \mathcal{M}, \mathcal{N}_c$)
2:      $\mathcal{I}_u \leftarrow \varnothing$
3:      while $\mathcal{N}_c > 0$ do
4:          $\mathcal{I} \leftarrow \mathcal{L}(\mathcal{M}[\mathcal{P}])$
5:          $b, \_, \_ \leftarrow \mathcal{O}(\mathcal{P}, \mathcal{I})$
6:          if $b$ then return Success $\mathcal{I}$
7:          else
8:              $\mathcal{I}_u \leftarrow \mathcal{I}_u \cup \mathcal{I}$
9:              $\mathcal{N}_c \leftarrow \mathcal{N}_c - 1$
10:     return HOUDINI($\mathcal{P}, \mathcal{I}_u$)
```

Fig. 5: Inference algorithm with Houdini

```
1:  procedure HOUDINI($\mathcal{P}$, $\mathcal{I}$)
2:      while $\mathcal{I} \neq \varnothing$ do
3:          $b, \mathcal{I}_S, \mathcal{I}_{NI} \leftarrow \mathcal{O}(\mathcal{P}, \mathcal{I})$
4:          if $b$ then return Success $\mathcal{I}$
5:          if $\mathcal{I}_S \neq \varnothing$ then $\mathcal{I} \leftarrow \mathcal{I} - \mathcal{I}_S$
6:          else
7:              if $\mathcal{I}_{NI} = \varnothing$ then return Failure
8:              else $\mathcal{I} \leftarrow \mathcal{I} - I_L$
9:      return Failure
10:
```

Fig. 6: Houdini algorithm

an adaptation of the Houdini algorithm to our setting. The algorithm takes as input a program $\mathcal{P}$ and a set of candidate invariants $\mathcal{I}$. It either returns Success $\mathcal{I}_I$, where $\mathcal{I}_I \subseteq \mathcal{I}$ and $\bigwedge_{i \in \mathcal{I}_I} i$ is an inductive loop invariant strong enough to verify $\mathcal{P}$, or it returns a Failure if it cannot find such a subset. The algorithm repeatedly queries the oracle with its current set of candidate invariants $\mathcal{I}$. Recall that the output of oracle is a triple $(b, \mathcal{I}_S, \mathcal{I}_{NI})$, where b is a boolean, $\mathcal{I}_S$ is the set of syntactically invalid candidates, and $\mathcal{I}_{NI}$ is the set of non-inductive candidates.

If the oracle returns true, the algorithm returns with Success $\mathcal{I}$. Otherwise, it removes one or more candidates from the set $\mathcal{I}$ and repeats the process. This pruning happens in one of two ways. If there are some candidate invariants that are syntactically invalid, they are pruned away. If there are no syntax errors, and the set $\mathcal{I}_{NI}$ is empty, the procedure returns Failure: this indicates the case when the current set $\mathcal{I}$ is a valid inductive invariant, but still not sufficient (i.e., strong enough) to verify the program. Otherwise the candidates in $\mathcal{I}_{NI}$ are pruned away and the loop repeats. The soundness of the Houdini algorithm follows directly from the soundness of the oracle. Houdini returns Success $\mathcal{I}$ only when the oracle verifies $\mathcal{P}$ with $\mathcal{I}$. Furthermore, the algorithm makes a linear number of calls to the oracle (linear in the size of candidate invariant set $\mathcal{I}$). In addition to soundness, Houdini guarantees to find the largest inductive subset of invariants [15].

Evaluation: Figure 4 (solid lines) show the impact of Houdini with both the prompt templates $\mathcal{M}_0$ and $\mathcal{M}_1$. With Houdini, the success rate for $k < 15$ is computed as an average over randomly sampling k out of the 15 completions, taking their union, and running Houdini.

Houdini has a significant positive impact. With 15 completions and the $\mathcal{M}_1$ prompt, the use of Houdini increases the success rate by **30.7%** (from 293 to 383 solved benchmarks). As before, the prompt $\mathcal{M}_1$ does better than $\mathcal{M}_0$ (383 to

```
1: procedure INFERENCE($\mathcal{P}, \mathcal{M}, \mathcal{N}_c, \mathcal{N}_r$)
2:     . . .
3:     $r \leftarrow$ HOUDINI($\mathcal{P}, \mathcal{I}_u$)
4:     if $r =$ Success _ then return $r$
5:     else return REPAIR($\mathcal{P}, \mathcal{I}_u, \mathcal{N}_r$)
```

Fig. 7: Inference algorithm with Repair

```
1: procedure REPAIR($\mathcal{P}, \mathcal{I}, \mathcal{N}_r$)
2:     $\_, \mathcal{I}_S, \mathcal{I}_{NI} \leftarrow \mathcal{O}(\mathcal{P}, \mathcal{I})$
3:     while $\mathcal{N}_r > 0$ do
4:         $\mathcal{I} \leftarrow \mathcal{L}(\mathcal{M}_r[\mathcal{P}, \mathcal{I}, \mathcal{I}_S, \mathcal{I}_{NI}])$
5:         $b, \mathcal{I}_S, \mathcal{I}_{NI} \leftarrow \mathcal{O}(\mathcal{P}, \mathcal{I})$
6:         if $b$ then return Success $\mathcal{I}$
7:         else
8:             $r \leftarrow$ HOUDINI($\mathcal{P}, \mathcal{I}$)
9:             if $r =$ Success _ then return $r$
10:            else $\mathcal{N}_r \leftarrow \mathcal{N}_r - 1$
11:    return Failure
```

Fig. 8: Repair algorithm

327 solved benchmarks). The results suggest that augmenting LLMs with symbolic techniques such as Houdini can increase the effectiveness of LLMs in solving such problems.

With the candidate invariants generated using prompt $\mathcal{M}_1$, we invoke Frama-C with timeout values higher than 3 seconds. We observed that the number of benchmarks verified by Frama-C remained the same with a timeout of 5 seconds and even 10 seconds.

F. *Using LLMs to repair incorrect invariants*

We explore using LLMs to repair the incorrect invariants, guided by the error messages produced by the oracle. This is motivated by an observation that in some cases minor changes to the LLM output can give us the correct invariants.

We parameterize our inference algorithm with another parameter $\mathcal{N}_r$ that denotes the maximum number of repair retries that the algorithm can make (Figure 7). Instead of returning the result of Houdini, as in Figure 5, the revised algorithm checks whether Houdini succeeds. If it does, the algorithm returns the result. If Houdini fails, it invokes a repair procedure.

The Repair algorithm, shown in Figure 8, takes as input the program $\mathcal{P}$, the set of all the LLM output invariants $\mathcal{I}_u$ across all completions, and the number of repair retries $\mathcal{N}_r$. It uses a specialized prompt template $\mathcal{M}_r$, templated over $\mathcal{P}$, a set of invariants $\mathcal{I}$, and $\mathcal{I}_S$ and $\mathcal{I}_{NI}$, incorrect subsets of $\mathcal{I}$ as returned by the oracle. The prompt template provides instructions to the LLM to repair the incorrect invariants. We show a snippet of $\mathcal{M}_r$ below:

```
Frama–C returns the following message:
{{ error }}

If the error message indicates a syntax error in the loop annotation,
fix the line with the syntax error. To fix the non–inductive
invariants, try the following:

If an invariant is preserved but not established, add a clause to the
invariant to make it established (a clause that makes the invariant
hold before the loop begins).

If an invariant is established but not preserved, add a clause to the
invariant to make it preserved (a clause that makes the invariant
hold after the loop ends, assuming that it holds before the loop
begins).

If an invariant is neither established nor preserved, remove it or
replace it with a different inductive invariant. If none of the
above is possible, add a new loop invariant to strengthen the
existing invariants.
```

The repair algorithm first invokes the oracle to get the errors $\mathcal{I}_S$ and $\mathcal{I}_{NI}$. It then prompts the LLM using $\mathcal{M}_r$, instantiated with $\mathcal{P}, \mathcal{I}, \mathcal{I}_S$, and $\mathcal{I}_{NI}$, to repair $\mathcal{I}$; the output of LLM is a new set of candidate invariants. The algorithm then uses the oracle and the Houdini procedure to find a sufficiently strong inductive set of invariants within the new set. The process repeats until either the algorithm succeeds in finding such a set or it runs out of the retries budget $\mathcal{N}_r$. The soundness of Repair follows from the soundness of the oracle and Houdini—it returns Success only when either the oracle or Houdini returns Success.

Evaluation: To evaluate our inference algorithm with Repair, we need to provide the $\mathcal{N}_r$ parameter. To keep the LLM budget the same as before, we make $\mathcal{N}_c + \mathcal{N}_r = 15$ so that the use of Repair does not increase the number of LLM queries. Observing that without Repair, the number of verified benchmarks starts to plateau at around 8 completions (Figure 4), we set $\mathcal{N}_c = 8$ and $\mathcal{N}_r = 7$.

With the repair procedure, LOOPY is able to verify 15 more benchmarks than before, bringing the number of benchmarks verified to 398/469. An example where repair helps is as shown. Before repair, the candidate invariants are `y == 10 - (x - 1)` and `y < 10`, both of which capture the behavior of `x` and `y` after the first iteration of the loop. The invariants, however, do not hold at the beginning of the first iteration when `y` is unconstrained. With Repair algorithm, the invariants are repaired to $\texttt{x == 1} \vee \texttt{y == 10 - (x - 1)}$ and $\texttt{x == 1} \vee \texttt{y < 10}$, allowing `y` to take any value before the first iteration. With these invariants, the program verifies.

```
void main()
{
  int x = 1;
  int y;
  while (x <= 10) {
    y = 10 - x;
    x = x + 1;
  }
  assert (y < 10);
}
```

G. *Comparing different LLMs*

To compare different LLMs, we evalute our inference algorithm, with and without Houdini, on two other models: GPT-3.5-Turbo and CodeLlama-34b-Instruct [10]. For this experiment, we fix the number of completions to 15 and use the prompt template $\mathcal{M}_1$. Figure 9 shows the results, we also plot the previously shown results for GPT-4 for comparison.

GPT-4 shows superior performance compared to the other models, although GPT-3.5-Turbo is a close second with 370

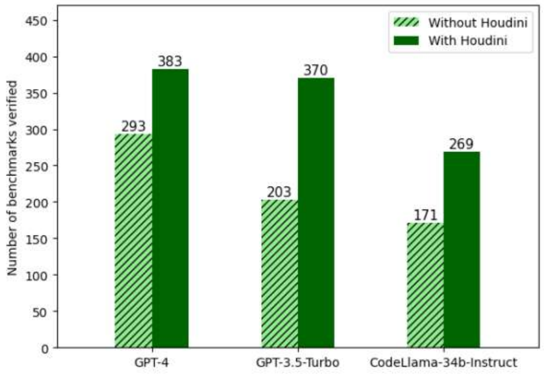


Fig. 9: LLMs comparison (with $\mathcal{M}_1$ and $\mathcal{N}_c = 15$)

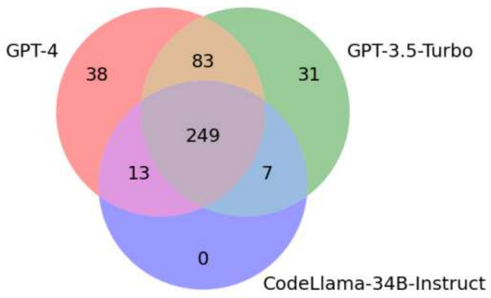


Fig. 10: Verified benchmarks

solved benchmarks when using Houdini. Interestingly, using Houdini helps other models "catch up" with GPT-4 by significantly increasing their success rates.

Figure 10 shows the intersection among the benchmarks verified using the different LLMs. GPT-4 has the most number of exclusively-solved benchmarks (38). GPT-3.5-Turbo is able to solve 31 benchmarks that GPT-4 could not. This experiment suggests that using multiple LLMs can help solve more benchmarks.

H. *Qualitative analysis*

We manually analyzed the failure cases for LOOPY and observed that for 10 failures, LOOPY is able to produce a correct and sufficient loop invariant, but Frama-C fails to verify the program. This implies a success rate of **408/469** for LLMs, augmented with our techniques. Among the successfully verified benchmarks, LOOPY generated, on average, 4.2 invariants per benchmark. Each of these invariants had, on average, 1.8 variables, and 2.1 operators (boolean, relational, and arithmetic), indicating that a fair number of invariants were non-trivial.

Analysis of failed benchmarks: We analyzed the benchmarks that LOOPY was not able to solve. For the 10 benchmarks for which LOOPY produces the right invariant but Frama-C fails to verify the program, we believe that it should be possible to strengthen Frama-C (e.g., one failure was due to missing axiomatization of integer mod operation). For the remaining 61 benchmarks, we manually came up with *an*

```
void main() {
  int x = 0, y = 0, flag = 0;
  while (flag < 1) {
    if (y < 0) flag = 1;
    if (flag < 1) x = x + 1;
    if (x < 50) y = y + 1;
    else y = y - 1;
  }
  assert(y == -2 && x == 99)
}
```
	```
	int main() {
	  int x = 0, y = 0, N;
	  if (N < 0)
	    return 1;

	  while (1) {
	    if (x <= N) y++;
	    else if (x >= N + 1)
	      y--;
	    else return 1;

	    if (y < 0) break;
	    x++;
	  }

	  if (N >= 0)
	    if (y == -1)
	      if (x >= 2 * N + 3)
	        assert(\false);

	  return 1;
	}
	```

Fig. 11: Example (a) where LOOPY fails (left), and (b) where Ultimate fails but LOOPY succeeds (right).

invariant that makes the program verify with Frama-C. Based on these ground truth invariants, we do a subjective classification of the failures into 4 categories. The classification is not indicative of features that are beyond LLMs today; there are also benchmarks in each of these categories that LLMs are able to solve.

The first category of failures are benchmarks that require disjunctions in the invariant. These benchmarks can be described as either having loops with multiple *phases* (i.e., as the loop iterates, the code path taken inside the loop changes several times, based on some flags or other branches), or assertions that depend on whether the loop is executed at all (needing a disjunct to account for the case when the loop is not entered at all). The program shown in Figure 11(a) has a multi-phase loop. It starts with x and y both at 0; then both increase by 1 in each iteration until x reaches 50, after which x continues to increase by 1 while y starts to decrease. Describing these "phases" requires one clause for each phase, connected by disjunction. We classified 44/61 failures in this category.

The second category of failures are benchmarks whose ground truth invariants requires a clause with at least three variables. An example of an invariant in this category is the following: $(0 < p) \wedge (2*q + r \leq w) \wedge (p == r + 2*i)$. 5/61 failures fall in this category.

The third category of failures are benchmarks where more precise constraints were required, compared to what was generated by our algorithm. For instance, for one of the benchmarks, the algorithm inferred the invariant $(k == x + y + z) \wedge (x \leq y) \wedge (y == z)$, which turned out to be an inductive invariant, but insufficient to prove the assertion. Changing the second clause to $x == y$ would make it work. There are 9/61 such failures. The fourth category, containing 3/61 failures, requires reasoning about floating-point arithmetic. It was hard to us, even manually, to come up with their ground-truth invariants.

Symbolic baseline: We compare the performance of our LLM-based inference algorithm the Ultimate tool [25] on the 469 benchmarks. Ultimate has higher success rate than Loopy with **430/469** benchmarks solved. However, we find that there are 31 benchmarks that Ultimate does not solve, but Loopy can solve. There are 63 benchmarks that Ultimate solves but Loopy does not. Ultimate and Loopy combined can solve **461/469** benchmarks, hinting that combining symbolic tools with LLM-based techniques can improve the existing state-of-the-art. Figure 11(b) shows an example from these 31 benchmarks. The assertion in the program can be verified with the loop invariant $(\mathsf{x} \leq \mathsf{N+1} \Rightarrow \mathsf{y} == \mathsf{x}) \wedge (\mathsf{x} > \mathsf{N+1} \Rightarrow \mathsf{y} == 2*(\mathsf{N+1}) - \mathsf{x})$, which Loopy infers but Ultimate does not.

To compare the run times of Loopy and Ultimate, we randomly selected 50 benchmarks from our dataset and measured the average time taken by each tool to verify a benchmark. In the case of Loopy, we generate 15 completions and check all of them with Frama-C. If all completions fail, then we run the Houdini loop. The average run time of Ultimate was 23.11s, and that of Loopy was 186.05s (including the LLM inference time which was 119.86s, using an unoptimized LLM-inference stack). Although optimizations to the Loopy implementation and the LLM inference stack could improve the run time, it is not in the scope of this work and we leave it as interesting avenues for future work.

I. Loop invariant inference for programs with arrays

We next evaluate our loop invariant inference algorithm on 169 benchmarks that use arrays. We use the algorithm shown in Figure 5, i.e. with Houdini but no repair, with 8 completions ($\mathcal{N}_c = 8$). With the prompt template $\mathcal{M}_0$, our algorithm is able to solve **60/169** benchmarks, while using the template $\mathcal{M}_1$ increases this number to **102/169** benchmarks.

On manual inspection of the failures, we find that LLM sometimes misses clauses that are common in the invariants for loops that manipulate arrays. To help LLMs in such cases, we add some array-specific instructions to the prompt. Some sample instructions are shown below:

```
For all the values and array ranges that do not change in the loop,
add an invariant equating them to their value before the loop. Add
a loop assigns clause listing all array ranges and variables
assigned for every loop. When a loop assigns an array,
–Invariants must specify state of all array elements after every
iteration of the loop, even if they have not changed yet.
–For the range of elements yet to be assigned by the loop, just
equate them to their value before the loop. For nested loops, use
the invariants of the inner loop as hints for the outer loop.
```

The instructions are mostly about capturing the precise state of the arrays in the invariants. With these instructions the number of solved benchmarks increases to **127/169**. Interestingly, these 127 are not a superset of the 102 solved with just $\mathcal{M}_1$ alone; Figure 12 shows a Venn diagram of the three sets: benchmarks solved with $\mathcal{M}_0$, with $\mathcal{M}_1$, and with $\mathcal{M}_1$ and instructions. Consider the array loop invariant example from

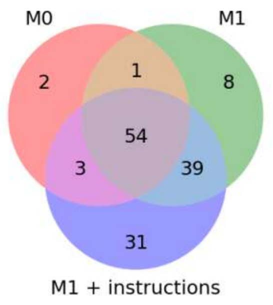


Fig. 12: Performance of Loopy instantiated with different prompts for arrays, $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_1$ + Instructions

Figure 1. The first for loop initializes the array a (of length N) s.t. a[i] = 3. The second loop, depending on whether N % (i + 1) is 0, either assigns a[i] = a[i] − 1 or a[i] = a[i] − 2. The third loop asserts that a[i] $\leq$ 2. With $\mathcal{M}_1$, the LLM fails to come up with the invariant \forall j. i <= j < N ==> a[j] == \at(a[j], Entry) for the second loop, stating that the values of array elements a[i] onwards are what they were at the beginning of the loop. This invariant is crucial for verifying the assert, and so, the program fails to verify. However, once we instruct the LLM to capture all array elements in the invariant, LLM outputs this clause, and the program verifies.

For the 42 failure cases, there are 9 benchmarks where Loopy infers the correct and sufficient loop invariants, but Frama-C fails to verify the program. Further, there are some programs where the loop invariants inferred by Loopy are close to the required invariants. In one of the benchmarks, for example, the loops iterate from 1 to N, but Loopy infers invariants where the index variable ranges from 0 to N. We believe such cases may be handled by using Repair (Figure 8); we leave this for future work.

IV. Program Termination

The problem: A ranking function is used to prove termination of a loop. In its simplest form, a ranking function V is an expression involving the variables used in the loop with the following two properties: (a) at the beginning of every loop iteration, the value of V is ≥ 0, and (b) the value of V strictly decreases with each loop iteration. Thus, the value of V at the beginning of an iteration provides an upper bound on the number of remaining loop iterations. A ranking function is sometimes also called a variant. There is a rich literature on algorithms for synthesizing ranking functions using abstract intepretation [31] constraint solving, model checking [32], [33] and more recently using custom trained neural networks [34]. We evaluate LLM capabilities to add to this body of work.

Beyond a simple expression, there are other common forms of ranking functions, lexicographic ranking functions and multi-phase ranking functions [35], [32]. A lexicographic ranking function is a ordered list $[V_i]$, where (a) at the beginning of a loop iteration, for all i, the value of V_i is ≥ 0, and (b) in every loop iteration, there exists j s.t. the value of V_j strictly decreases and $\forall k.\ k < j$, the value of V_k remains the

Prompts used	No. of benchmarks verified
M_2	133
M_2, M_3	170
M_2, M_3, M_4	181

Fig. 13: Results for ranking function inference

```
1: procedure VARIANTINFERENCE($\mathcal{P}, \mathcal{M}, \mathcal{N}_V, \mathcal{N}_I$)
2:     while $0 < \mathcal{N}_V$ do
3:         $\mathcal{V} \leftarrow \mathcal{L}(\mathcal{M}[\mathcal{P}])$
4:         $\mathcal{I} \leftarrow \varnothing$
5:         $n \leftarrow \mathcal{N}_I$
6:         while $0 < n$ do
7:             $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{L}(\mathcal{M}_I[\mathcal{V}, \mathcal{P}])$
8:             $n \leftarrow n - 1$
9:         $r \leftarrow$ Houdini($\mathcal{P}, \mathcal{I}$)
10:        if $r$ = Failure then $\mathcal{N}_V \leftarrow \mathcal{N}_V - 1$
11:        else
12:            Success $\mathcal{I} \leftarrow r$
13:            $r \leftarrow \mathcal{O}_\mathcal{T}(\mathcal{P}, \mathcal{V}, \mathcal{I})$
14:            if $r$ then return Success ($\mathcal{V}, \mathcal{I}$)
15:            else $\mathcal{N}_V \leftarrow \mathcal{N}_V - 1$
16:    return Failure
```

Fig. 14: Ranking function inference

same. A multi-phase ranking function [36] is a special case of lexicographic ranking function. It is an ordered list $[V_i]$, where when the loop execution starts, first V_1 decreases until it becomes non-positive, then V_2 decreases until it becomes non-positive, and so on. There are other forms of ranking functions and well-founded relations [37], [38], [39], [40], [36], [35], but we restrict focus to only the ones described above.

To prove that a ranking function is valid for a loop, one might need additional loop invariants to establish key properties about the loop. The problem, therefore, is to infer both a ranking function as well as the supporting invariants that are needed to prove its correctness.

We assume access to an oracle $\mathcal{O}_\mathcal{T}$ that takes as input a program $\mathcal{P}$ (with a single loop), a (candidate) ranking function $\mathcal{V}$, and an inductive loop invariant $\mathcal{I}$. It returns a boolean value where true implies that $\mathcal{V}$ can be proven to be a valid ranking function using $\mathcal{I}$. Frama-C provides such an interface, and we use it as our oracle.

Ranking function inference algorithm: Figure 14 shows our ranking function inference algorithm. It takes as input a program $\mathcal{P}$ with a single loop, a prompt template $\mathcal{M}$, and two number of completions parameters $\mathcal{N}_V$ and $\mathcal{N}_I$. It returns either Success $(\mathcal{V}, \mathcal{I})$, where $\mathcal{V}$ and $\mathcal{I}$ are ranking function and inductive loop invariant for the loop in $\mathcal{P}$ respectively, or Failure otherwise.

The algorithm instantiates the prompt template $\mathcal{M}$ with $\mathcal{P}$, and queries the LLM to infer a candidate ranking function $\mathcal{V}$. To be able to invoke the oracle to check the LLM output, we need an inductive loop invariant as well. The algorithm infers it using the techniques developed in the last section. Specifically, it instantiates a prompt template $\mathcal{M}_I$ with the candidate ranking function $\mathcal{V}$ and the program $\mathcal{P}$, and prompts the LLM. Template $\mathcal{M}_I$ instructs the LLM to infer an inductive loop invariant required for proving a given ranking function. The algorithm collects the LLM output loop invariants for $\mathcal{N}_I$ completions, and invokes the Houdini algorithm (Figure 6). If Houdini succeeds, the algorithm invokes the oracle $\mathcal{O}_\mathcal{T}$ with $\mathcal{V}$ and $\mathcal{I}$ (the output of Houdini). If the oracle returns true, the algorithm returns Success $(\mathcal{V}, \mathcal{I})$. Whereas if either Houdini or the oracle fails, the algorithm repeats until it succeeds or it exhausts the maximum number of retries $\mathcal{N}_V$. The soundness of the algorithm follows from the soundness of Houdini (Figure 6) and the oracle.

Evaluation: We evaluate our ranking function inference algorithm on 281 benchmarks, with one method and one loop each. We set the parameters $\mathcal{N}_V$ and $\mathcal{N}_I$ to 5 each, and as mentioned before, use Frama-C as the oracle. We show that by adding increasingly domain-specific instructions to the prompt template $\mathcal{M}$, we can solve more benchmarks.

The prompt template $\mathcal{M}_I$ contains instructions for the LLM to infer inductive loop invariants for a given ranking function. The prompt mentions that the LLM should infer an invariant that implies the ranking function decreases with every iteration. It also contains the loop invariant instructions similar to those in the M_1 prompt from the previous section. The full prompt is available in the public repository[1].

We first evaluate a basic prompt template for inferring the loop ranking function. The prompt template explains to the LLM what a ranking function is, but it does not instruct the LLM to infer a specific kind of ranking function (lexicographic or multi-phase, for instance). The complete prompt text is available in the public repository[1]. The result of using this prompt template is shown in Table 13 as the prompt M_2. As can be seen, with this prompt, our algorithm is able to solve **133/281** benchmarks.

On a closer inspection of the failures, we find that while the algorithm could solve simple cases of ranking functions (e.g., when it is a single expression), it did not do so well on benchmarks that require lexicographic or multi-phase ranking functions. We, next, add instructions for inferring lexicographic ranking functions.

```
A lexicographic ranking function is a sequence of expressions with
the property that each expression must be positive for the loop to
execute. For example, if (e1, e2, e3) is a lexicographic ranking
function, then with each loop iteration, either e1 is positive and
decreases, or e1 remains the same and e2 is positive and decreases
or e1 and e2 remain the same and e3 is positive and decreases.
Find a lexicographic ranking function for the loop in the
following program.
```

With this prompt, the algorithm is able to solve **37** more benchmarks (prompt M_3 in Table 13). Finally, we try similar instructions for the multi-phase ranking functions, and solve **11** more benchmarks (prompt M_4 in Table 13), taking the total verified benchmarks to **181/281**. The example shown in Figure 1(c) is one of the benchmarks that fails with the basic prompt, but with the multi-phase prompt, our algorithm infers `[z; y; x]` as a multi-phase loop ranking function.

Symbolic baseline: Ultimate solves **236/281** benchmarks. There are **162/281** that both our algorithm and Ultimate solve,

```
// @requires n >= 0;
// @ensures \result == n % 2;
int isOdd(int n) {
    if (n == 0)  return 0;
    else if (n == 1) return 1;
    else return isEven(n - 1);
}
// @requires n >= 0;
// @ensures \result == 1 - n % 2;
int isEven(int n) {
    if (n == 0) return 1;
    else if (n == 1) return 0;
    else return isOdd(n - 1);
}
int main() {
    int n = unknown_int();
    if (n < 0) return 0;
    int result = isOdd(n);
    assert(result >=0 && result != n % 2);
}
```

Fig. 15: Example recursive program along with LOOPY-generated annotations. The annotated program verifies with Frama-C.

19/281 that only our algorithm solves, and **74/281** that only Ultimate solves. Thus our algorithm and Ultimate combined solve **255/281**.

V. Recursive programs

While the primary focus of this work has been on dealing with the complexity of loops, we also explore the ability of LLMs to deal with recursive programs. Specifically, programs where the methods are (mutually-) recursive; Figure 15 shows an example. The task here is to infer the pre- and postconditions for the methods in the program, such that Frama-C is able to verify the assertions in the program. The prompt we used for this task is available in the public repository[1]. With a total 31 benchmarks, LOOPY is able to successfully verify $16/31$programs with 8 completions. Ultimate is able to verify 20/31 of these benchmarks. Figure 15 also shows the pre- and postconditions inferred by LOOPY.

VI. Related work

LLMs for invariant generation: Pei et al. [41] study this problem by building dataset of programs and corresponding invariants and then fine-tune a pre-trained LLM on this dataset. Our approach does not rely on fine-tuning and directly evaluates the capabilities of foundational models. Furthermore, Pei et al. do not focus on generating *inductive* invariants that are necessary for establishing a formal proof of correctness.

Lemur [42] presents a proof calculus and an algorithm to use an LLM to generate and repair invariants. Lemur uses a symbolic verifier to check for inductiveness and generate counterexamples. They use a chaining approach to iteratively strengthen, repair or backtrack on proposed invariants. This necessitates a proof of soundness for single method with loops, and may require further extensions for an interprocedural setting. It is unclear if Lemur would find an inductive invariant even if the LLM proposes all of its ingredients, since Lemur is sensitive to the order in which invariants are proposed. Integrating Houdini with Lemur could be a promising direction for future work. Further, their approach does not apply to proving termination, and even for invariants, it has been evaluated on a much smaller set of benchmarks. Lemur is publicly available but we were unable to run it. On the benchmarks of the Lemur paper, LOOPY and Lemur perform comparably when given the same budget of LLM queries. Among the 133 Code2Inv benchmarks, LOOPY solves 103 benchmarks while Lemur solves 107 benchmarks. Among Lemur's 50 SV-COMP benchmarks, both tools solve 26 benchmarks each.

Yao et al. [43] leverage LLMs to semi-automate proofs for Rust programs in the context of the Verus program verifier. However, they do not consider loop termination or working across multiple methods. Furthermore, their evaluation is not fully automated for the benchmark and only compared against a purely manual baseline. Chakraborty et al. [44] build iRank, a custom model for ranking candidate invariants. iRank is orthogonal and complementary to our work; we can use it as a heuristic for decreasing the number of calls to the oracle by only checking highly-ranked invariant candidates.

LLMs for proof assistants: LLMs have been used to automate proof synthesis in interactive proof assistants [45], [46]. Leandojo [46], for instance, fine-tunes a retrieval model for lemma selection and a generative model for proof generation for the Lean theorem prover. Given the general purpose nature of these proof assistants, these approaches are not tailored for automatic program verification that we target in this paper. Our approach also does not require fine-tuning a model.

A. Threats to validity

A potential concern while working with LLMs is the problem of data contamination, which happens when the benchmarks used for evaluation were already a part of the training data used for the models. In this case, the models can overfit, which affects their ability to generalize to newer benchmarks.

There is no ideal way to completely remove contamination while working with industrial models, or even for open-source ones, given the scope of training data that they consume. We compensate, to the best of our ability, by considering as many benchmarks as we could try, and increasing the diversity of tasks as well as program and invariant features (arrays, ranking functions, etc.). The Stack [47], which is a public code corpus used to train open source models like StarCoder [48] and DeepSeek-Coder-V2 [49], does not contain the SVCOMP and Code2Inv benchmarks. We are also not aware of any other data source where these programs appear alongside their invariants.

Another concern is about the reproducibility of our results. Closed models, such as GPT-4, can get updated any time, which can affect the numbers reported in this paper. Our toolchain is parametric on the choice of LLMs and we do use an open-source model (CodeLlama) to compensate for this concern. LLMs are also stochastic, implying that they can return different responses for the same query. Using multiple completions helps compensate for this stochasticity.

References

[1] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 42–56. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908099

[2] M. Brockschmidt, Y. Chen, P. Kohli, S. Krishna, and D. Tarlow, "Learning shape analysis," in *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, ser. Lecture Notes in Computer Science, F. Ranzato, Ed., vol. 10422. Springer, 2017, pp. 66–87. [Online]. Available: https://doi.org/10.1007/978-3-319-66706-5_4

[3] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 499–512. [Online]. Available: https://doi.org/10.1145/2837614.2837664

[4] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 106–120.

[5] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "Cln2inv: Learning loop invariants with continuous logic networks," in *International Conference on Learning Representations*, 2020.

[6] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[7] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, "Code2inv: A deep learning framework for program verification," *Computer Aided Verification*, vol. 12225, pp. 151 – 164, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211027794

[8] OpenAI, "GPT-4 technical report," https://doi.org/10.48550/arXiv.2303.08774, 2023.

[9] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, and et al., "Palm 2 technical report," *CoRR*, vol. abs/2305.10403, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.10403

[10] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, and et al., "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[11] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, and et al., "Training language models to follow instructions with human feedback," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[12] GitHub, "Github copilot," https://github.com/features/copilot, 2022.

[13] Amazon, "Amazon codewhisperer," https://aws.amazon.com/codewhisperer/, 2023.

[14] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0326-7

[15] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, ser. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 500–517.

[16] OpenAI, "GPT-3.5," https://platform.openai.com/docs/models/gpt-3-5, 2023.

[17] J. Signoles, B. Desloges, and K. Vorobyov, *E-ACSL User Manual*. [Online]. Available: http://frama-c.com/download/e-acsl/e-acsl-manual.pdf

[18] K. Madhukar, B. Wachter, D. Kroening, M. Lewis, and M. Srivas, "Accelerating invariant generation," in *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '15. Austin, Texas: FMCAD Inc, 2015, p. 105–111.

[19] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven chc solver," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 707–721. [Online]. Available: https://doi.org/10.1145/3192366.3192416

[20] I. Dillig, T. Dillig, and A. Aiken, "Fluid updates: Beyond strong vs. weak updates," in *Programming Languages and Systems*, A. D. Gordon, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 246–266.

[21] S. Chakraborty, A. Gupta, and D. Unadkat, "Diffy: Inductive reasoning of array programs using difference invariants," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 911–935.

[22] D. Beyer, "Competition on software verification and witness validation: Sv-comp 2023," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Nature Switzerland, 2023, pp. 495–522.

[23] The Termination Competition, "The Termination Problem Database," https://github.com/TermCOMP/TPDB, 2023.

[24] X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li, "Large-scale analysis of non-termination bugs in real-world oss projects," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 256–268. [Online]. Available: https://doi.org/10.1145/3540250.3549129

[25] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, ser. Lecture Notes in Computer Science, J. Palsberg and Z. Su, Eds., vol. 5673. Springer, 2009, pp. 69–85.

[26] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[27] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[28] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-Ergo 2.2," in *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, Jul. 2018. [Online]. Available: https://inria.hal.science/hal-01960203

[29] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[31] P. Cousot and R. Cousot, "An abstract interpretation framework for termination," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, J. Field and M. Hicks, Eds. ACM, 2012, pp. 245–258. [Online]. Available: https://doi.org/10.1145/2103656.2103687

[32] C. Urban, A. Gurfinkel, and T. Kahsai, "Synthesizing ranking functions from bits and pieces," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 54–70.

[33] M. Brockschmidt, B. Cook, and C. Fuhs, "Better termination proving through cooperation," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 413–429. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_28

[34] M. Giacobbe, D. Kroening, and J. Parsert, "Neural termination analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 633–645. [Online]. Available: https://doi.org/10.1145/3540250.3549120

[35] J. Leike and M. Heizmann, "Ranking Templates for Linear Loops," *Logical Methods in Computer Science*, vol. Volume 11, Issue 1, Mar. 2015. [Online]. Available: http://lmcs.episciences.org/797

[36] A. M. Ben-Amram and S. Genaim, "On multiphase-linear ranking functions," *CoRR*, vol. abs/1703.07547, 2017. [Online]. Available: http://arxiv.org/abs/1703.07547

[37] M. Colón and H. Sipma, "Synthesis of linear ranking functions," in *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, ser. Lecture Notes in Computer Science, T. Margaria and W. Yi, Eds., vol. 2031. Springer, 2001, pp. 67–81. [Online]. Available: https://doi.org/10.1007/3-540-45319-9_6

[38] A. R. Bradley, Z. Manna, and H. B. Sipma, "Linear ranking with reachability," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 491–504. [Online]. Available: https://doi.org/10.1007/11513988_48

[39] ——, "The polyranking principle," in *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, ser. Lecture Notes in Computer Science, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds., vol. 3580. Springer, 2005, pp. 1349–1361. [Online]. Available: https://doi.org/10.1007/11523468_109

[40] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 239–251. [Online]. Available: https://doi.org/10.1007/978-3-540-24622-0_20

[41] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23, 2023.

[42] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," 2023.

[43] J. Yao, Z. Zhou, W. Chen, and W. Cui, "Leveraging large language models for automated proof synthesis in rust," 2023.

[44] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, "Ranking llm-generated loop invariants for program verification," in *Findings of The 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP-findings 2023)*, 2023.

[45] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," 2023.

[46] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," 2023.

[47] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[48] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[49] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers

Daniel Mendoza
Stanford University
Stanford, CA, USA
dmendo@stanford.edu

Christopher Hahn[†]
X, the moonshot factory
Mountain View, CA, USA
chrishahn@google.com

Caroline Trippel
Stanford University
Stanford, CA, USA
trippel@stanford.edu

***Abstract*—Automating translation from natural language (NL) to temporal logic (TL) specifications offers to democratize verification of hardware systems. However, this vision is challenged by (i) inherent ambiguity in NL, which can create multiple plausible translation possibilities, and (ii) the need to validate translation output, when the translator can make mistakes. To address these challenges, we propose SYNTHTL, an interactive approach and tool, which uses large language models (LLMs), model checkers, and oracle (human) guidance, to translate an NL specification of a hardware design's intended behavior into a TL specification that reflects the NL and holds (formally) on the design.**

SYNTHTL performs *structured translation*, whereby it first decomposes a complex unstructured NL specification into a logical combination of simple NL sub-specifications. Then, it produces TL translations of the simple NL sub-specifications, called *sub-translations*, and mechanically combines these sub-translations to yield a TL translation of the complex NL specification. This approach significantly reduces the oracle effort required to validate the translation output, since only sub-translations and the logical relationships between them need to be inspected. Plus, it enables the design of automated model checker utilities, which efficiently guide the search for a translation that holds on the design, despite inherent NL ambiguity. With SYNTHTL, we conduct the largest LLM-assisted NL to TL specification translation case study to date, producing a correct formalization of the Arm AMBA AHB bus protocol.

I. Introduction

Hardware verification involves checking that a *Design Under Test* (DUT) upholds desired properties through simulation, formal model checking, and/or runtime verification [1]. These properties, or *specifications*, are typically expressed as *Temporal Logic* (TL) formulas (e.g., using Linear Temporal Logic (LTL) [2], SystemVerilog Assertions (SVAs) [3], or Property Specification Language (PSL) [4] syntax), which formally define sets of allowed execution traces on the DUT. Today, verification experts *manually* derive TL specifications for a DUT from (hardware designer-friendly) *natural language* (NL) specifications. Consequently, the time and resources required to produce thorough design-specific TL specifications limits the application of verification in practice [5].

The NL to TL specification translation bottleneck is a natural target for *natural language processing* (NLP) approaches (e.g., *Large Language Models*, or LLMs), which have recently been deployed to resolve it [6], [7], [8], [9], [10], [11], [12], [13]. Yet, NLP is not a panacea. First, producing TL specifications that capture the intent of NL is challenged by its inherent ambiguity; a single NL phrase may represent *many* plausible TL formulae. Second, NLP approaches are susceptible to producing blatantly incorrect outputs.

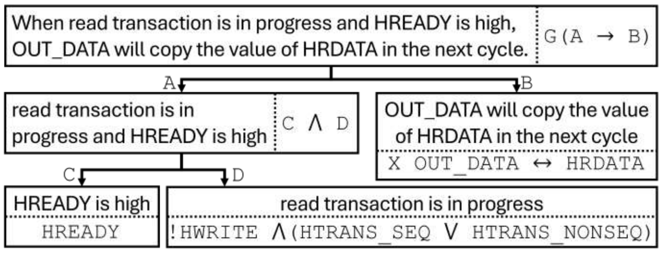

When read transaction is in progress and HREADY is high, OUT_DATA will copy the value of HRDATA in the next cycle.
G(A → B)
A
B
read transaction is in progress and HREADY is high
C ∧ D
OUT_DATA will copy the value of HRDATA in the next cycle
X OUT_DATA ↔ HRDATA
C
D
HREADY is high
HREADY
read transaction is in progress
!HWRITE ∧(HTRANS_SEQ ∨ HTRANS_NONSEQ)

Fig. 1: *Sub-translation tree* that mechanically composes to produce LTL specification: $G((\texttt{HREADY}\wedge !\texttt{HWRITE}\wedge(\texttt{HTRANS\_SEQ} \vee \texttt{HTRANS\_NONSEQ}))\rightarrow X(\texttt{OUT\_DATA}\leftrightarrow \texttt{HRDATA}))$. Nodes (*sub-translations*) consist of an *NL sub-specification* and a corresponding *TL sub-specification*. Directed edges from a parent node to its children denote a decomposition.

For these reasons, NLP-based NL to TL specification translation can produce an abundance of candidate outputs, especially when specifications are complex. For instance, GPT4 [14] generates over 100K unique TL specification possibilities when translating an NL specification of the *arbiter* component of the Arm AMBA AHB bus protocol [15], [16] to TL in our case study (§V). Existing NLP-based TL formalization approaches thus rely on a human user to manually validate final generated outputs, i.e., *full* TL specifications [6], [7], [8], [9], [10], [11], [12], [13]. Unfortunately though, validating full output specifications can be just as difficult as writing them from scratch.

A. This Paper

We propose SYNTHTL, an approach and tool that uses LLMs, model checkers, and oracle (human) guidance, to translate an NL specification of some DUT's desired behavior into a TL specification that reflects the NL and holds (formally) on the DUT. SYNTHTL leverages LLMs to perform *structured translation of unstructured NL*, whereby it first decomposes a complex unstructured input *NL specification* (i.e., an NL specification that does not conform to a predefined

[†]Work done while at Stanford University.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_17

grammar) into a logical combination of simple *NL sub-specifications*. Then, it translates each NL sub-specification to a *TL sub-specification* and mechanically combines all TL sub-specifications according to their logical structure to yield a complete output *TL specification*.

SYNTHTL's structured translation procedure can be visualized as a *sub-translation tree* (Figure 1). Nodes represent *sub-translations*, consisting of an NL sub-specification and a TL sub-specification, and labeled directed edges denote a decomposition of a parent sub-translation into a logical combination of simpler child sub-translations. The NL sub-specification of a sub-translation tree's root node is the (full) NL specification input to SYNTHTL, from which the tree is recursively generated using LLMs (with optional oracle input) as follows. First, SYNTHTL introduces zero or more fresh symbols to represent (serve as "placeholders" for) unique strict substrings in a parent node's NL sub-specification. Then, it generates the parent node's TL sub-specification as a TL formula over these symbols. Finally, for each symbol, it instantiates a child node, whose incoming edge is labeled with the symbol and whose NL sub-specification is the substring that the symbol represents.

SYNTHTL expects the oracle to inspect each node of a complete sub-translation tree to validate that its TL sub-specification is a reasonable translation of its NL sub-specification and that its child nodes exhibit a reasonable decomposition. Inaccuracies are corrected by the oracle, possibly with the help of LLMs or other external tooling. Upon oracle sign-off, SYNTHTL uses model checkers to evaluate whether the full tree's corresponding (mechanically generated) TL specification holds on the DUT. If it does, the NL specification, TL specification, and DUT are all deemed correct, since oracle sign-off (earlier) confirms the TL specification is consistent with the NL specification. If it does not, there is a bug in the (consistent) NL/TL specifications and/or the DUT, warranting further oracle investigation.

Our primary insight is that SYNTHTL's structured translation approach drastically reduces overall oracle effort. First, when validating a sub-translation tree, an oracle need only inspect simple sub-translations (nodes) and their immediate decompositions (children) and never the end-to-end translation (i.e., full TL specification). Second, model checkers can leverage this tree structure to guide the oracle towards an NL/TL specification or DUT fix when oracle tree validation or model checker TL specification evaluation fails.

SYNTHTL offers two such model checker-guided utilities: *culprit identification* and *translation search*. When an output TL specification does not hold on the DUT, culprit identification uses model checkers to find sub-translations (from its sub-translation tree) that logically contribute to the failing output specification. The oracle can prioritize fixing these sub-translations (i.e., NL/TL sub-specifications), their decompositions, or the DUT, as appropriate. To account for inherent ambiguity in NL and/or inaccurate sub-translations, translation search enables the oracle to provide (manually or using an LLM) multiple decomposition options and/or TL sub-specification options per tree node, yielding *set* of possible sub-translation trees (syntactically unique TL specifications) to be systematically checked against the DUT.

Unsurprisingly, SYNTHTL's translation search utility often produces a set of unique sub-translation trees that is too large to model check exhaustively. However, **our secondary insight** is that SYNTHTL's structured translation can be exploited to make this analysis practical. First, SYNTHTL's translation search utility is designed to detect inconsistent sub-trees, which cannot be extended to a full TL specification that holds on the DUT; its *sub-tree pruning* optimization discards all sub-translations trees that contain these sub-trees to avoid redundantly checking them on the DUT. Second, translation search uses a *batch model checking* optimization, which identifies all unique conjunctive clauses among the remaining (not pruned) full translations that must be checked on the DUT and queries a model checker at most once for each.

Overall, this paper significantly eases the burden of NL to TL specification translation via the following contributions:

- **SYNTHTL Approach & Tool:** We propose SYNTHTL to translate NL to TL specifications that both represent the intent of the NL and hold on some target DUT, using LLMs, model checkers, and oracle guidance.
- **Structured Translation:** SYNTHTL decomposes a (complex) NL to TL translation problem into a set of simpler, mechanically-composable sub-problems, organized as a sub-translation tree, that are easier to solve and validate.
- **Model Checker-Guided Translation:** SYNTHTL uses model checkers to guide an oracle towards a correct NL to TL translation by flagging potential culprit sub-translations when an output TL specification does not hold on the DUT and efficiently uncovering a correct translation out of a large space of possible options.
- **Case Study:** We use SYNTHTL to translate three real-world NL specifications—comprising the Arm AMBA AHB bus protocol [15], [16]—to LTL. Our evaluation features much larger NL specifications (maximum/average of 643/449 words) and TL specifications (maximum/average of 490/434 symbols, i.e., LTL operators and variables) than prior work, which focuses on individual NL properties within a larger specification. For comparison, each NL specification in the recent `nl2spec` dataset [6] contains a maximum/average 21/10 words; TL specifications contain a maximum/average of 37/10 symbols. Among 7.26e16 LLM-generated candidate TL specifications for an NL specification of the *controller* component of the AMBA AHB protocol, SYNTHTL converges to a correct one, consisting of 56 sub-translations. In doing so, SYNTHTL queries an oracle to validate 96 sub-translations and to fix 18/11 TL sub-specification/decompositions, where the average TL formula fixed by the oracle is 3% the size of the full correct TL specification.

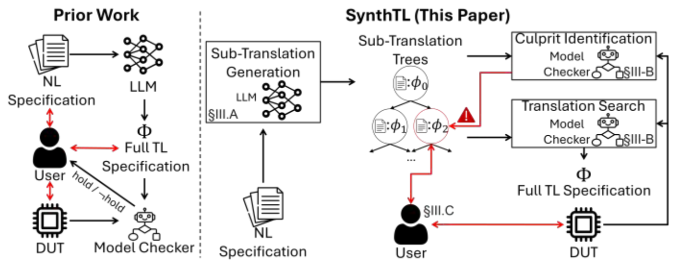


Fig. 2: Existing NL to TL specification translation approaches (left) rely on manual validation of full TL specifications. SYNTHTL (right) reduces manual effort using sub-translation trees and model checker guidance.

II. Related Work and Motivation

In this section, we give an overview of prior work on translating structured (§II-A) and unstructured (§II-B) NL to TL specifications in order to motivate SYNTHTL (§II-C), which translates unstructured NL to TL.

A. Structured Translation of Structured NL

Early work on NL to TL specification translation requires *structured* NL as input, i.e., NL that conforms to a pre-defined grammar [17], [18], [19], [20]. By *mechanically* deriving an output TL specification from a structured NL input, these approaches conduct *structured translation*, similar to SYNTHTL once it has produced a sub-translation tree. The output TL specification is guaranteed to be correct if the input NL specification is. However, supplying structured NL inputs can be burdensome.

B. Unstructured Translation of Unstructured NL

Recent NLP advances have inspired numerous efforts to translate *unstructured* NL (i.e., no grammar restrictions) to TL specifications, e.g., using LLMs [6], [8], [9], [11], [12], [10] and other NLP methods [7], [13]. Unlike SYNTHTL, these approaches conduct *unstructured translation*, meaning that an LLM or NLP algorithm ultimately constructs the final TL specification output in one shot (Figure 2, left).

Translating unstructured NL is challenged by its inherent ambiguity, e.g., the phrase "signal READY holds" can be formulated in various plausible ways, like the following in LTL: READY, XREADY, GREADY, READY $\leftrightarrow$ XREADY. Plus, NLP is prone to making translation mistakes. To resolve both issues, prior work [6], [7], [8], [9], [10], [11], [12], [13] expects a user to manually validate the final formalization output, confirming that it captures the intent of the NL input.

Unfortunately, this validation step can be just as difficult as manually performing the entire NL to TL specification translation task. Moreover, when a translated TL specification is deemed inconsistent with the DUT (e.g., by a model checker), localizing bugs in the input NL specification, output TL specification, and/or DUT is difficult, as is fixing them.

We categorize prior work on unstructured translation of NL to TL specifications based on whether translation is conducted *end-to-end* (§II-B1) or *interactively* (§II-B2).

1) End-to-end Translation: End-to-end approaches [10], [11], [8], [12], [13], [9] deploy specialized prompting or training schemes to elicit accurate NL to TL specification translations from an NLP model without soliciting user input beyond the NL specification itself.

2) Interactive Translation: Interactive approaches, like LTLtalk [7] and `nl2spec` [6], query the user to validate/fix candidate translations before the final output is produced.C

LTLtalk [7] queries a user to accept or reject sample traces from full candidate TL specifications. But, manually analyzing traces of a complex TL specification is challenging.

Similar to SYNTHTL, `nl2spec` [6] decomposes the translation task into easier sub-translations and queries a human oracle to fix sub-translations. However, since no structure is enforced among sub-translations, `nl2spec` requires an LLM to compose them into a full TL specification in one shot. Plus, this lack of structure means that formal model checking cannot be applied to automatically localize errors or fix particular sub-translations.

C. Our Approach: Structured Translation of Unstructured NL

We propose *structured translation of unstructured NL* to TL specifications with SYNTHTL (Figure 2, right).

Without loss of generality, we focus our discussion on translating NL to LTL [2], which extends propositional logic with temporal operators including U (until), X (next), F (eventually), and G (globally). Figure 1 illustrates a property from the AMBA AHB bus protocol specification [15], [16] formalized in LTL. Operators can be nested (e.g., $GF\phi$ means ϕ occurs infinitely often).

SYNTHTL uses LLMs, model checkers, and oracle guidance to iteratively transform an input unstructured NL specification into a logical combination of sub-translations, which are mechanically combined to yield an output TL specification. Thus, the oracle (user) need only manually validate (simpler) sub-translations and their logical organization, but never full TL specifications. Plus, this logical organization enables SYNTHTL to localize inconsistencies between an output TL specification and the DUT to sub-translations and efficiently guide the translation procedure towards bug fixes.

III. SynthTL Approach and Tool: Structured Translation of Unstructured Natural Language to Temporal Logics

We present the SYNTHTL approach and tool in this section and detail its model checker utilities in §IV. Figure 3 illustrates how SYNTHTL translates NL phrases from the AMBA AHB bus protocol specification [15], [16] into an LTL specification.

A. Interactive TL Specification Generation

Given an input NL specification, SYNTHTL performs automated sub-translation tree generation using LLMs and queries an oracle to validate the resulting tree(s). A *sub-translation*

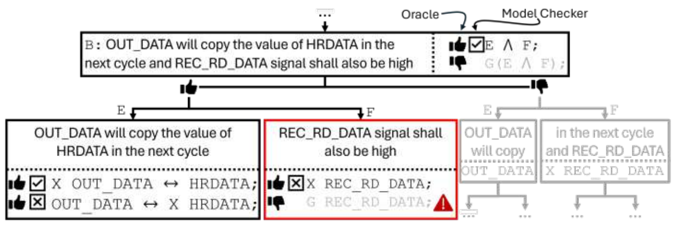

Oracle
Model Checker
B: OUT_DATA will copy the value of HRDATA in the next cycle and REC_RD_DATA signal shall also be high
E ∧ F;
G(E ∧ F);
E
F
OUT_DATA will copy the value of HRDATA in the next cycle
X OUT_DATA ↔ HRDATA;
OUT_DATA ↔ X HRDATA;
REC_RD_DATA signal shall also be high
X REC_RD_DATA;
G REC_RD_DATA;
OUT_DATA will copy
OUT_DATA
in the next cycle and REC_RD_DATA
X REC_RD_DATA

Fig. 3: For each NL sub-specification, SYNTHTL generates multiple possible decompositions and TL sub-specifications. An oracle may accept/reject (thumbs up/down) any of them. To determine if a sub-translation tree's TL specification holds on the DUT, SYNTHTL checks if each of its TL specification holds or not (checks or "x"s). If no tree holds, SYNTHTL flags sub-translations as potential root causes (red nodes).

tree (e.g., Figure 1) represents a single complete translation from an input NL specification to an output TL specification. Each node is a sub-translation, which consists of an NL sub-specification and a corresponding TL sub-specification. Labeled edges denote a decomposition of a parent sub-translation into a logical combination of simpler child sub-translations.

1) Sub-Translation Tree Generation and Validation: In sub-translation tree generation, SYNTHTL first instantiates a sub-translation tree root node whose NL sub-specification is populated with the input NL specification. It then uses LLMs (with optional user guidance) to generate a complete tree by recursively decomposing and translating nodes, starting with the root.

Decomposing a parent node involves first introducing zero or more fresh symbols, each of which serves as a representative for a unique strict substring in the parent's NL sub-specification. For each symbol, a child node is instantiated, with the symbol labeling the directed edge from parent to child. For example, in Figure 1, A labels the edge from the root node to its left child and represents the substring "read transaction is in progress and HREADY is high." The parent node's TL sub-specification is then produced by translating its NL sub-specification under the mapping of its substrings to their representative symbols.

In practice, when supplied with an input NL specification, SYNTHTL generates from it a *set* of sub-translation trees. In particular, for each parent node, it generates up to D unique decomposition options and K unique TL sub-specification options per decomposition; K and D are hyperparameters, which specify the number of LLM queries. While SYNTHTL deploys LLMs to conduct this step, it discards decompositions either where the symbols do not represent unique strict substrings within the parent's NL sub-specification or where one symbol represents a substring of another symbol (i.e., redundant symbols). SYNTHTL also discards TL sub-specifications that: are not well-formed, are trivial (i.e., $\top$ or $\bot$), do not use all symbols that label edges to its node's children, or use symbols beyond those that represent its node's children and variables defined in the DUT.

Following each node decomposition and TL sub-specification generation step during sub-translation tree generation, SYNTHTL queries the oracle to accept/reject each decomposition and TL sub-specification option (thumbs up/down in Figure 3). If ever the oracle rejects all decompositions or TL sub-specifications for a node, SYNTHTL asks the oracle to provide acceptable ones. For each accepted decomposition, fresh decomposition, translation, and validation steps are initiated for its newly-instantiated children.

Recursion terminates when encountering a node with no decomposition (i.e., zero symbols/child nodes). A node cannot be decomposed if the only strict substring that can be derived from its NL sub-specification is the empty string. In practice, LLM-based decomposition may terminate before this condition is reached. For example, in the sub-translation tree in Figure 1, the NL sub-specification "HREADY is high" has no decomposition.

2) Structured Translation of Sub-Translation Trees to TL Specifications: Sub-translation tree generation and validation produces a set of candidate sub-translation trees, which can be obtained by performing a cross product among all validated decomposition and TL sub-specification options per node. The TL specifications implied by these candidate sub-translation trees can be derived recursively as follows. Starting at their root nodes, replace placeholder symbols in each parent node's TL sub-specification with the TL sub-specifications of the symbols' corresponding children. For example, in Figure 1, C $\wedge$ D will be transformed into HREADY $\wedge$ (!HWRITE $\wedge$ (HTRANS_SEQ $\vee$ HTRANS_NONSEQ)).

Note that given a sub-translation tree with N nodes, D decomposition options per node, K TL sub-specification options per node, the number of candidate sub-translation trees is $O((KD)^N)$. However, validating all generated trees only requires the oracle to validate decompositions and TL sub-specifications for each node, and thus the number of times the oracle validates a decomposition or TL sub-specification is $O(KDN)$ (significantly lower than all possible trees).

3) LLM Prompts for Sub-Translation Tree Generation: SYNTHTL queries an LLM during sub-translation tree generation to decompose NL sub-specifications and to perform NL to TL sub-specification translation. Our prototype implementation of SYNTHTL relies on *in-context* learning [21] with distinct prompting strategies for each task.

For NL sub-specification decomposition, an LLM is prompted to output a JSON dictionary, which maps variables to substrings of the input NL sub-specification. The prompt includes a description of the decomposition task and examples of correct decompositions.

For NL to TL sub-specification translation (i.e., to produce a sub-translation), SYNTHTL prompts an LLM to generate a TL sub-specification that uses the variables introduced in the decomposition of its corresponding NL sub-specification. The prompt presents examples of correct sub-translations to the LLM. To encourage the LLM to generate TL sub-specifications that use variables in the DUT, SYNTHTL includes a list of all DUT variable names in the prompt. We also observe that NL context required to correctly translate an individual NL sub-specification is often scattered across the large input NL specification that contains it (as a substring). To handle such situations, SYNTHTL uses *retrieval augmented generation* (RAG) [22] to extract relevant context for an NL sub-specification within an input NL specification using a retrieval model. This context is prepended to the TL sub-specification generation prompt.

4) Sub-Translation Tree Expressiveness: Note that SYNTHTL extracts structure from unstructured NL to make translation easier, but it retains full expressiveness of the unstructured input NL and structured output TL. A sub-translation tree can be understood as overlaying the inductive structure of a well-formed TL formula on top of an NL specification. If no structure can be extracted from the input NL specification, the result is a tree with just the root node (i.e., no decomposition), where the node's corresponding TL sub-specification is the full TL specification. However, we did not encounter such a non-decomposable NL specification in our evaluation (§V). Since TL sub-specifications are TL sub-formulae, and sub-translation trees recursively define a top-level TL formula as logical combination of TL sub-specifications using TL operators, SYNTHTL retains the full expressiveness of the TL.

B. Searching for Translations that Hold on DUT

After sub-translation tree generation and validation (§III-A1), SYNTHTL deploys a translation search procedure to find a translation, or more concretely an output TL specification, that holds on the DUT out of an exponential number of candidates (§III-A2). If some TL specification holds on the DUT, the DUT upholds the NL specification's requirements, since the oracle previously confirmed that the TL specification was a reasonable interpretation of the NL specification (by validating all sub-translations and decompositions, §III-A1). If some TL specification does not hold on the DUT, one of two things could be true: the NL and TL specifications are consistent (due to oracle sign-off), and there is a bug in the DUT; or the NL and TL specifications are inconsistent (due to ambiguity that lead to an errant oracle sign-off). In the latter case, there may or may not be a bug DUT as well.

Given a set of candidate sub-translation trees and their TL specifications, one could query a model checker to determine which hold on the DUT. However, exhaustively checking $O((KD)^N)$ TL specifications (§III-A2) is computationally expensive (and likely infeasible). SYNTHTL's model checker-assisted *translation search* utility (§IV-A) addresses this issue in two ways. First, it leverages *sub-tree pruning* (§IV-A1) to discard many candidate sub-translation trees, which cannot hold on the DUT due to inconsistent sub-trees. Second, when checking DUT adherence to the TL specifications that are not pruned, translation search leverages *batch model checking* (§IV-A2) to ensure that common conjunctive clauses among these specifications are checked at most once. Note that if multiple TL specifications hold on the DUT, SYNTHTL returns only the most constrained (i.e., most restrictive) ones

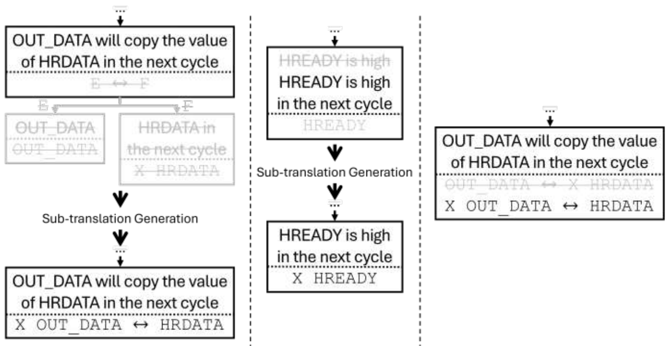

OUT_DATA will copy the value of HRDATA in the next cycle
~~E ↔ F~~
E
F
~~OUT_DATA~~
~~OUT_DATA~~
~~HRDATA in the next cycle~~
~~X HRDATA~~
Sub-translation Generation
OUT_DATA will copy the value of HRDATA in the next cycle
X OUT_DATA ↔ HRDATA
~~HREADY is high~~
HREADY is high in the next cycle
~~HREADY~~
Sub-translation Generation
HREADY is high in the next cycle
X HREADY
OUT_DATA will copy the value of HRDATA in the next cycle
~~OUT_DATA ↔ X HRDATA~~
X OUT_DATA ↔ HRDATA

Fig. 4: Example of how SYNTHTL's users may fix a culprit node's decompositions (left), NL sub-specification (middle), or TL sub-specifications (right). After editing a decomposition or NL sub-specification, SYNTHTL returns to its sub-translation generation and validation (§III-A1) step to generate new decompositions and TL sub-specifications for the modified node. This is followed by translation search (and culprit identification if needed) (§III-B). After editing a TL sub-specification or the DUT, SYNTHTL returns to its translation search step.

by default; however, it can be configured to return all TL specifications that hold.

If no candidate sub-translation tree produces a TL specification that holds on the DUT, SYNTHTL's model checker-assisted *culprit identification* utility (§IV-B) outputs a set of the least constrained trees/specifications among them. Rather than require the user to manually analyze each failing specification, culprit identification flags a subset of the nodes in their corresponding sub-translation trees, which may be relevant for the inconsistency (red nodes in Figure 3).

C. Fixing Culprit Sub-Translations or the DUT

Once culprit nodes have been identified (§III-B), the oracle can elect to fix the DUT, the NL/TL sub-specifications within culprit nodes, and/or decompositions of culprit nodes. After applying a fix, the SYNTHTL procedure returns back to and repeats an earlier analysis phase as follows. If the oracle modifies an NL sub-specification or decomposition of a culprit node, SYNTHTL goes back to its sub-translation and validation step (§III-A1) to interactively (with the oracle) propagate the these changes by re-generating TL sub-specifications and decompositions for edited nodes. Figure 4 demonstrates examples for which a decomposition is modified (left) and NL sub-specification is modified (middle). If the oracle modifies the DUT or a TL sub-specification, SYNTHTL returns back to its translation search step (§III-B) to check if the edits give rise to a full TL specification that holds on the DUT. Figure 4 illustrates an example in which a TL sub-specification is modified (right).

Algorithm 1 Translation Search Algorithm

```
1: function TSEARCH($n, S_{IN}$, mode)
2:     res = {}
3:     for decomposition in getNodeDcmps($n, S_{IN}$) do
4:         $n$.setChildren(decomposition)
5:         if mode = LC then                    ▷ get TL sub-specifications for $n$
6:             subTLSet = getLCNodeSubTL($n, S_{IN}$)   ▷ get LC TL for $n$
7:         else
8:             subTLSet = getAllNodeSubTL($n, S_{IN}$)   ▷ get all TL for $n$
9:         for subTLSpec in subTLSet do
10:            $n$.setSubTL(subTLSpec)
11:            if mode ≠ LC then             ▷ Check least constrained trees first
12:                $C_n = \{T \mid T \in S_{IN} \land t_n \in T\}$   ▷ trees that contain $t_n$
13:                $C$ = TSEARCH(Root($t_n$), $C_n$, $LC$)        ▷ get LC trees
14:                $C_{\text{DUT}} = \{T \mid T \in C \land T \models \text{DUT}\}$   ▷ check LC trees
15:            if mode = LC ∨ $C_{\text{DUT}} \neq \emptyset$ then   ▷ Recurse to children of $n$
16:                cList = [TSEARCH(child, $S_{IN}$, mode) for child in $n$]
17:                for $c1, c2, \ldots$ in crossProduct(cList) do
18:                    $n'$ = copy($n$).setChildren($c1, c2, \ldots$)
19:                    res.add($n'$)
20:    return res
```

Algorithm 2 Batch Model Checking Algorithm

```
1: function BATCHMC($tSet$, DUT)
2:     $H_{\text{clause}}$ = hashTable() ▷ Initialize hash table, mapping clause to trees
3:     for $T$ in $tSet$ do
4:         for clause in getCNF($T$) do
5:             $H_{\text{clause}}$[clause].add($T$)   ▷ Add all CNF clauses to hash table
6:     while $|H_{\text{clause}}| > 0$ do            ▷ Loop while clauses left to check
7:         $D_{\text{clause}} = \{\}$                 ▷ Clauses checked in current iteration
8:         $D_T = \{\}$   ▷ Trees that do not hold on DUT in current iteration
9:         for clause in $H_{\text{clause}}$ do
10:            $D_{\text{clause}}$.add(clause)
11:            if clause $\not\models$ DUT then        ▷ Check if clause holds on DUT
12:                $D_T = H_{\text{clause}}$[clause]         ▷ found inconsistent clause
13:                break
14:        for clause in $H_{\text{clause}}$ do            ▷ Update trees in hash table
15:            $H_{\text{clause}}$[clause] = $H_{\text{clause}}$[clause] − $D_T$
16:            if $|H_{\text{clause}}[\text{clause}]| == 0$ then
17:                $D_{\text{clause}}$.add(clause)  ▷ Remove clause if all trees not hold
18:        $tSet = tSet - D_T$          ▷ Remove trees inconsistent with DUT
19:        $H_{\text{clause}} = H_{\text{clause}} - D_{\text{clause}}$     ▷ Remove clauses already checked
20:    return $tSet$
```

IV. SYNTHTL UTILITIES: MODEL CHECKER-GUIDED TRANSLATION SEARCH AND CULPRIT IDENTIFICATION

We now provide more details on SYNTHTL's model checker-guided utilities: translation search (§IV-A), which features sub-tree pruning (§IV-A1) and batch model checking (§IV-A2), and culprit identification (§IV-B).

A. Translation Search Utility

SYNTHTL's translation search procedure takes as input a set of sub-translation trees. Let $S_{IN} = \{T_1, T_2, ...\}$ denote the input set of trees, each of which represents a TL specification (i.e., a TL formula). Given S_{IN}, translation search outputs the set $S_{OUT} \subseteq S_{IN}$. If one or more input TL specifications hold on the DUT, S_{OUT} contains the *most constrained* TL specifications that hold on the DUT. If none hold (i.e., S_{OUT} is empty), SYNTHTL employs culprit identification (§IV-B). Note that there can be multiple most/least constrained TL specifications, because in general, TL specifications may not be strict supersets or subsets of one another.

Concretely, translation search returns:

$S_{\text{OUT}} = \{T \mid T \in S_{IN} \land T \models \text{DUT} \land \forall i, T_i \in S_{IN} \land (T \neq T_i \rightarrow T \not\models T_i)\}$

1) Sub-Tree Pruning in Translation Search: SYNTHTL's translation search coordinates the instantiation of candidate sub-translation trees that result from sub-translation tree generation and validation (§III-A1). To obtain the set of all candidate sub-translation trees, one could naively take the cross product of all possible decompositions and TL sub-specifications (§III-A2). However, translation search avoids exhaustively constructing and checking all TL specifications by incrementally constructing these candidate trees (i.e., incrementally constructing the cross product) node by node starting at the root and preemptively pruning candidate trees before they are fully generated. The pseudocode for this procedure, called TSEARCH, is shown in Algorithm 1 and detailed below. Translation search uses TSEARCH to obtain a pruned set of possible TL specifications, and then checks them against the DUT to obtain S_{OUT}.

TSEARCH starts from a root node n, input tree set S_{IN}, and mode $\neq LC$; the mode variable indicates whether TSEARCH should output all sub-translation trees that may hold on the DUT (mode $\neq LC$), or the least constrained (LC) set of trees among all possible (mode $= LC$). TSEARCH constructs all decompositions (line 3) and all TL sub-specifications (line 9) of node n and conditionally recurses to each of its child nodes (line 15). Given the recursion condition is true, the set of all possible trees that contain node n is obtained through a cross product of each possible child sub-tree of node n (§III-A2, line 16 - 19).

Suppose node n has just been instantiated, and let t_n denote a partially constructed sub-tree up to and including n. TSEARCH only recurses to n's child nodes along a particular decomposition if it is possible to extend sub-tree t_n to a TL specification that holds on the DUT (lines 11 - 14). TSEARCH determines the recursion condition by constructing all of the least constrained (mode $= LC$) TL specifications that extend sub-tree t_n (lines 12 - 13), and model checking them against the DUT (line 14). If none of these least constrained TL specifications hold on the DUT, then no other trees that extend sub-tree t_n can hold, so TSEARCH can prune them from consideration (and avoid explicitly checking them against the DUT) by terminating recursion.

Note that constructing the set of least constrained TL specifications with sub-tree t_n (lines 12 - 13) does not require exhaustively constructing all trees with sub-tree t_n since TSEARCH with mode $= LC$ (line 13) incrementally constructs least constrained trees through only considering the least constrained TL sub-specifications for each node (line 6). Also, our implementation of Algorithm 1 makes use of memoizing results to avoid redundant recursive calls (not shown).

2) Batch Model Checking: After obtaining the pruned set of sub-translation trees, SYNTHTL deploys a novel batch model checking utility to improve performance of identifying TL

Algorithm 3 Conjunctive Clause Extraction Algorithm

```
1: function EXTRACTCLAUSES($T, n$)
2:     $n$.setChildren($\{\}$)
3:     $n$.setSubTL(id)
4:     $\Phi_T$ = GetTLSpec($T$)
5:     conjSet = getCNF($\Phi_T$)
6:     return $\{c \mid c \in$ conjSet if id $\in c\}$
```

specifications that hold on the DUT. Batch model checking, as shown in Algorithm 2, takes as input a DUT, and a set of sub-translation trees $tSet$, and efficiently obtains the subset of these sub-translation trees that hold on the DUT. Batch model checking exploits common conjunctive clauses among the set of TL specifications, so that each conjunctive clause is checked at most once across all trees.

First, conjunctive clauses for each sub-translation tree are discovered by transforming its corresponding TL specification into a conjunctive form (line 4). Second, to identify common conjunctive clauses among the trees in $tSet$, a hash table is used to map clauses to the sub-translation trees that contain them (lines 2 to 5). Third, as long as the hash table is non-empty, the algorithm iteratively selects clauses to check against the DUT with a model checker (lines 6 to 19). Whenever a clause is found to not hold on the DUT (line 11), all of its associated sub-translation trees and clauses are pruned from the hash table (line 14 to 19).

B. Culprit Identification Utility

Suppose that translation search (§III-B) determines that no TL specifications—from all those produced during sub-translation tree generation and validation (§III-A1)—hold on the DUT. At this point, SYNTHTL deploys culprit identification to identify nodes within the least constrained TL specifications (among all generated TL specifications) that are possible culprits for (i.e., possibly contributing to) their inconsistencies with the DUT. Culprit identification obtains the set of least constrained TL specification using TSEARCH (Algorithm 1, §IV-A) with mode $= LC$ starting from the root node, given the input set of all possible trees from sub-translation tree generation and validation. A node is a possible culprit if it contributes to at least one conjunctive clause in a full TL specification that does not hold on the DUT.

To identify all possible culprit nodes, within a sub-translation tree T that does not hold on the DUT, SYNTHTL first extracts for each node n of T the set of conjunctive clauses in the full TL specification that it contributes to using Algorithm 3. First, the decomposition of node n is set to empty (line 2) and its TL sub-specification is set to a special identifier id (line 3). Then, from this new variant of T, a full TL specification is constructed (line 4) and transformed into a conjunctive form (line 5). Finally, the clauses which contain id are returned (line 6). SYNTHTL then checks if each of the clauses returned by Algorithm 3 hold on the DUT.

This approach ensures that all nodes that are responsible for a TL specification's inconsistency with the DUT are flagged as possible culprits. However, not all nodes flagged as possible culprits are true positives, since non-culprits can contribute to failing clauses. To reduce false positives in culprit identification, SYNTHTL applies a heuristic filter to prioritize nodes which are more likely to be true culprits. Instead of flagging a node as a culprit if it contributes to at least one failing clause, the heuristic filter only marks a node as a culprit if all clauses it contributes to fail. Our evaluation demonstrates this heuristic significantly improves the precision of culprit identification (fewer false positives) and identifies true culprits while retaining high accuracy (few false negatives, §V-D).

V. CASE STUDY: TRANSLATING AN INDUSTRIAL NL SPECIFICATION TO TL WITH SYNTHTL

As a case study, we use SYNTHTL to translate three real-world NL specifications comprising the Arm AMBA AHB bus protocol [15], [16] to TL. In doing so, we evaluate SYNTHTL's sub-translation tree generation and validation (§III-A1), translation search (§IV-A), and culprit identification (§IV-B) components. Moreover, we answer the following questions: How successful are LLMs in generating sub-translation trees and to what degree do sub-translation trees reduce manual effort in validating TL specifications compared to existing methods (§V-A)? How efficient is translation search compared to exhaustive search (§V-B)? To what degree does culprit identification localize inconsistencies with the DUT to particular sub-translations (§V-D)?

Prototype Implementation Our SYNTHTL prototype consists of ~3K lines of Python code and queries the open-source LTL model checker, Spot [23]. Although our prototype implementation of SYNTHTL deals with LTL, the SYNTHTL approach can be used to generate formulas in other TLs (e.g., SVA [3], PSL [4], STL [24], and so on). We have published our code (including LLM prompts), the AMBA AHB benchmarks (taken from prior work [16]), and example generated outputs in a public repository.[1]

Benchmarks We evaluate SYNTHTL on three real-world NL hardware specifications, taken from the AMBA AHB bus protocol [15], [16], corresponding to its *arbiter*, *controller*, and *worker* modules. We select these NL specifications for our experiments for two reasons. First, AMBA AHB is an industrial protocol that is widely deployed in modern SoCs, including Arm Cortex-M based designs [25], [26]. Second, it has already been manually formalized in prior work [16], giving us "ground truth" TL specifications to use in qualitatively assessing SYNTHTL's outputs. The original TL specification is written in PSL [16], and we manually rewrite it in LTL since our prototype implementation of SYNTHTL uses LTL model checking.

For each of the three AMBA AHB hardware modules we consider, Table I gives the sizes of their published NL specifications and corresponding ground truth TL specifications [16]. These TL specifications serve the role of "golden" DUTs in our evaluation.

Large Language Models We conduct our evaluation with two state-of-the-art LLMs: GPT3.5 and GPT4 [14].

[1]https://github.com/dmmendo/SynthTL

Module	Controller	Worker	Arbiter
NL Size (words)	436	268	643
TL Size (symbols)	460	351	490

TABLE I: Specification sizes used in SYNTHTL's evaluation.

A. Evaluation of TL Generation and Validation

We compare SYNTHTL's sub-translation tree generation and validation (§III-A1) to `nl2spec` [6], the existing state-of-the-art approach, in terms of manual effort required to produce the correct TL specification.

Setup In this experiment, We query the LLM $D = K = 3$ times to generate decompositions and TL sub-specifications for each node with SYNTHTL. The oracle selects one correct TL sub-specification and one correct decomposition for each sub-translation with SYNTHTL and provides a correct TL sub-specification/decomposition if none of the LLM-generated options are correct.

Baseline Given an input NL specification, `nl2spec` [6] decomposes it into sub-translations and produces an output TL specification from the sub-translations, all using an LLM. The user can iteratively edit sub-translations and request that `nl2spec` re-generate (with an LLM) the output TL specification from them until the user decides the TL specification is correct.

Unlike SYNTHTL, `nl2spec`'s sub-translations are *unstructured* (i.e., they are not organized as a sub-translation tree), and so they cannot be mechanically composed into a full TL specification. Instead, `nl2spec` uses an LLM to perform this composition in one shot. However, this approach renders `nl2spec` susceptible to generating inaccurate TL specifications, even if the user has decided that all sub-translations are correct.

In terms of oracle effort, both `nl2spec` and SYNTHTL require sub-translation validation. However, `nl2spec` additionally requires oracle validation of full output TL specifications, while SYNTHTL additionally asks the oracle to validate (comparatively much simpler) sub-translation decompositions.

We consider an *ideal* `nl2spec` as our baseline, which is initially (in its first iteration) given a correct set of sub-translations taken from the leaf nodes of a correct "reference" SYNTHTL sub-translation tree. Thus, `nl2spec` need only compose these sub-translations to produce a correct full TL specification using an LLM. If the full TL specification generated by `nl2spec` in an iteration is incorrect, the oracle provides new sub-translations in the next iteration that compose the current sub-translations by instantiating them within the context of their parents' TL sub-specifications in the reference sub-translation tree. For example, consider a reference sub-translation tree representing TL specification $(A \wedge B) \rightarrow (C \wedge D)$ with two leaf nodes whose TL sub-specifications are $A \wedge B$ and $C \wedge D$. The TL sub-specification of the root node defines the implication ($\rightarrow$) between the leaf nodes. In the first iteration, `nl2spec` is given the leaf nodes. If `nl2spec` does not output the correct TL specification in the first iteration, the oracle provides $(A \wedge B) \rightarrow (C \wedge D)$ in the second iteration. Thus, the sub-translations provided by the oracle become fewer and more-complex/coarse-grained in each iteration of `nl2spec`, approaching the full reference TL specification. `nl2spec` continues iterating either until it generates the correct full TL specification by composing sub-translations with an LLM or until the oracle supplies the full TL specification as input (by composing sub-translations from the previous failed iteration). The LLM is given three tries (i.e., is queried three times) in each iteration. Incorrect sub-translations are discarded between iterations.

Metrics For both SYNTHTL and our ideal `nl2spec` baseline, we record the following:

- Number of unique generated full TL specifications (**Space Size**).
- Number of TL sub-specifications and decompositions inspected by the oracle (**Inspections**)
- Number of TL sub-specifications and decompositions edited by the oracle (**Trns Edit** and **Dcmp Edit**). For each node, the oracle only edits a decomposition or TL sub-specification if the LLM does not generate one that is deemed valid by the oracle.
- Number of nodes in final sub-translation tree (**Tree Size**).
- Size of formulas inspected (**Inspect Size**) and edited (**Edit Size**) by the oracle in number of symbols. A symbol is an LTL operator or variable. We normalize this quantity by the size of the ground truth TL specification (from the handwritten TL specification in prior work [16]).

Results The results are shown in Table II. Across all six SYNTHTL experiments (three AMBA AHB modules and two LLM options), SYNTHTL generates up to 9.33e17 and as few as 8.29e4 unique TL specifications. This result demonstrates that the input NL specifications for modules of the Arm AMBA AHB bus protocol are highly ambiguous, despite being carefully crafted so as to be amenable to formalization [16].

SYNTHTL produces a correct TL specification, with the oracle inspecting and editing TL sub-specifications that are on average 2.43% (up to 20.9%) and 2.95% (up to 8.9%) the size of the ground truth TL specification, respectively. Further, the oracle edits on average 12.9% (up to 20.6%) and 37.5% (up to 44.8%) the number of decompositions and TL sub-specifications, respectively, that exist in the final sub-translation tree. That is, most decompositions and sub-translations in the final correct sub-translation tree are automatically generates by the LLM. These results demonstrate that SYNTHTL requires significantly less manual effort than both manual end-to-end NL to TL translation and full TL specification validation.

In all six `nl2spec` experiments, our ideal `nl2spec` baseline fails to produce the correct TL specification. It incorrectly composes TL specifications despite being given correct the sub-translations in every iteration. In all cases, the size of the inspected and edited sub-specifications is significantly smaller with SYNTHTL compared to `nl2spec`. This result demonstrates that SYNTHTL enables LLM-based NL to TL translation to handle large and complex NL specifications for the first time. The maximum, average, and total size of

Arm AMBA AHB Module	Controller				Worker				Arbiter			
Large Language Model	GPT3.5		GPT4		GPT3.5		GPT4		GPT3.5		GPT4	
Translation Approach	SynthTL	nl2spec	SynthTL	nl2spec	SynthTL	nl2spec	SynthTL	nl2spec	SynthTL	nl2spec	SynthTL	nl2spec
Space Size	7.26e16	5	8.29e4	5	3.05e10	6	2.99e6	6	9.33e17	6	5.97e8	6
Inspections	96	125	92	77	94	90	81	66	107	90	117	94
Trns Edit	18	18	17	18	21	20	16	21	26	22	25	23
Dcmp Edit	11	18	12	18	2	20	0	21	8	22	12	23
Tree Size	56		58		48		47		58		61	
Sum Inspect Size	1.515	9.124	1.254	6.898	1.558	11.538	1.197	8.077	2.506	5.543	2.157	8.682
Avg Inspect Size	0.021	0.073	0.020	0.090	0.027	0.128	0.023	0.122	0.028	0.062	0.027	0.092
Max Inspect Size	0.209	1.020	0.150	1	0.142	1.302	0.202	1	0.127	1	0.120	1
Sum Edited Size	0.541	2.585	0.546	2.585	0.627	2.823	0.595	3	0.606	2.773	0.629	2.804
Avg Edited Size	0.030	0.144	0.032	0.144	0.030	0.141	0.037	0.143	0.023	0.126	0.025	0.122
Max Edited Size	0.089	1	0.089	1	0.177	1	0.177	1	0.084	1	0.084	1
Gen Correct?	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗

TABLE II: Generating and validating the AMBA AHB TL specification with SynthTL versus nl2spec. Formula size is normalized by the size of the full ground truth specification.

Module	Worker		Controller		Arbiter	
	K=2	K=3	K=2	K=3	K=2	K=3
Exhaustive	4096	5832	4096	3888	4096	2916
SynthTL	505	376	2052	804	548	184
% Pruned	87.7	93.6	49.9	79.3	86.6	93.7

TABLE III: Exhaustive vs. SynthTL's translation search

Module	Arbiter	Controller	Worker
Formulas	128	128	10000
Clauses	640	62	1512
Variables	19	29	24
Exhaustive (s)	28075.69	10025.26	462.38
SynthTL (s)	7622.44	2975.77	13.09
Speedup	3.68	3.37	35.32

TABLE IV: Batch vs. Exhaustive Model Checking

inspected TL sub-specifications is on average 5.32×, 3.94×, 6.97× smaller with SynthTL compared to nl2spec, respectively. The maximum, average, and total size of edited TL sub-specifications is on average 4.68×, 4.68×, 9.61× smaller with SynthTL compared to nl2spec, respectively. These results show that SynthTL's sub-translation trees significantly reduce the manual effort required to inspect and fix sub-translations compared to nl2spec.

Takeaway SynthTL significantly reduces manual effort required to fix and validate LLM-generated TL specifications compared to prior approaches and enables automatic generation of large and complex TL specifications.

B. Evaluation of Translation Search

Next, we evaluate the efficiency of translation search (§IV-A) in discovering a correct TL specification among a set of sub-translation trees, given a correctly implemented DUT (i.e., the ground truth TL specification produced in prior work [16]). To generate a set of sub-translation trees, we direct SynthTL to conduct sub-translation tree generation and validation (§III-A) and limit the oracle to specifying one correct decomposition and $K = 2, 3$ TL sub-specifications per node, where at least one TL sub-specification is correct. Note that querying the LLM multiple times for TL sub-specifications/decompositions may produce equivalent TL sub-specifications/decompositions and, in such situations, duplicates are discarded. To evaluate under multiple settings, we also limit the oracle to only provide multiple TL sub-specifications for a node if the tree space size would be less than 2^{13}.

Results Table III shows the number of TL specifications generated and checked on the DUT with SynthTL's translation search utility (SynthTL row) versus exhaustively searching all trees in the input set (Exhaustive row), i.e., the maximum number of TL specifications to check on the DUT. In all cases, translation search explores significantly fewer sub-translation trees than the exhaustive approach to find the correct TL specification. SynthTL prunes as many as 93.6%, 79.3%, 94.7% of the search space for the arbiter, controller, and worker, respectively.

Takeaway SynthTL's translation search greatly improves the scalability of model checking many translation possibilities for a given NL specification through effective sub-tree pruning.

C. Evaluation of Batch Model Checking

We now evaluate the efficacy of SynthTL's batch model checking optimization (§IV-A2) in accelerating model checking many TL specifications against a DUT.

Results Table IV shows runtimes of exhaustive and batch model checking for a set of sub-translation trees generated by an LLM with SynthTL's sub-translation generation. Batch model checking is 3.68×, 3.37×, 35.32× faster compared to exhaustive model checking for the arbiter, controller, and worker, respectively.

Takeaway Batch model checking significantly accelerates model checking large sets of TL specifications.

D. Evaluation of Culprit Identification

We now evaluate the effectiveness of SynthTL's culprit identification in localizing inconsistencies with the DUT to particular sub-translations (§IV-B) when a sub-translation tree contains an incorrect sub-translation (§V-D1), and when the DUT is incorrectly implemented (§V-D2). Note that AC refers to culprit identification with no heuristic filters, and FC refers to the approach with the heuristic filter (§IV-B).

Module	Controller						Worker						Arbiter					
Bug	G		X		$\neg$		G		X		$\neg$		G		X		$\neg$	
Ex.	13		35		28		16		41		53		11		23		30	
App.	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC
% Clpt	57.0	34.5	67.0	45.2	58.0	32.0	68.9	47.4	70.7	50.1	65.7	40.4	52.23	22.1	54.7	24.2	58.3	31.1
Recall	1	0.92	1	0.97	1	1	1	1	1	1	1	0.96	1	0.91	1	0.91	1	1

TABLE V: Culprit Identification given an incorrect AMBA AHB TL specification

Module	Controller						Worker						Arbiter					
Bug	G		X		$\neg$		G		X		$\neg$		G		X		$\neg$	
Ex.	3		8		10		16		42		50		2		9		10	
App.	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC
% Clpt	42.5	6.3	41.4	5.2	53.1	24.1	59.7	30.7	61.2	33.8	67.4	44.4	42.9	6.4	48.7	16.1	53.8	24.4
Recall	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

TABLE VI: Culprit Identification given an incorrect AMBA AHB DUT

1) Culprit Identification Given an Incorrect TL Specification and a Correct DUT: To emulate a sub-translation tree that contains an incorrect sub-translation that causes the full TL specification to not hold on a correctly implemented DUT, we insert bugs into a correct sub-translation tree. Given a correct sub-translation tree, we randomly select a node, and change the sub-specification by adding an operator (i.e., either G, X, or $\neg$). If the perturbation causes the TL specification to not hold on the DUT, then we run culprit identification and record the percentage of nodes flagged as possible culprits and if the set of possible culprits contains the true culprit.

Results Table V shows the number examples evaluated per perturbation, the average percentage of nodes that are marked as possible culprits per perturbation, and the fraction of examples where the possible culprit set contained the true culprit (recall). AC flags on average 56.8% of the nodes, and the identified possible culprit set always contains the true culprit (i.e., recall is always 1). FC flags 38.5% of the nodes (fewer than AC), however, due to its heuristic nature, catches the true culprit in 97.1% of all cases (fewer than AC).

In Figure 5, we show the percentage of nodes in the sub-translation tree that are marked as possible culprits versus the percentage of true culprits in the sub-translation tree when perturbing randomly selected sub-translations with incorrect LLM-generated sub-specifications. Note that the percentage of true culprits does not go to 100%, because the LLM-generated sub-specifications for the remaining set of nodes do not cause inconsistencies with the DUT. In all cases both AC and FC find all the true culprits. The percentage of nodes flagged as possible culprits increases with the percentage of true culprits in the sub-translation tree. AC flagged as few as 49.1%, 37.9%, 38.1% and as high as 90.6%, 96.6%, 92.1% for the controller, worker, and arbiter, respectively. FC flagged as few as 15.1%, 5.2%, 4.8% and as high as 90.6%, 96.6%, 92.1% for the arbiter, controller and worker, respectively.

2) Culprit Identification Given a Correct TL Specification and an Incorrect DUT: We evaluate the efficacy of culprit identification in localizing inconsistencies with the DUT to particular sub-translations, given a correct TL specification that does not hold on a buggy DUT. To emulate bugs in the DUT, we first construct a correct sub-translation tree for each module by manually transforming the ground truth LTL specification [16] into one. Then, we add an operator (i.e., either G, X, or $\neg$) in a randomly selected a sub-translation and use the perturbed TL specification as the DUT.

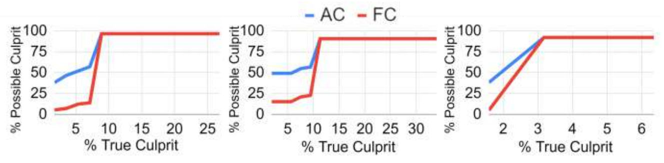


Fig. 5: Percent flagged possible culprits vs. true culprits of an AMBA AHB controller (left), worker (middle), arbiter (right).

Results Table VI shows what percent of nodes in the sub-translation tree are marked as possible culprits. Both AC and FC find all true culprits in all examples. AC/FC flag on average 59.7%/32.2% of nodes as possible culprits, respectively.

Takeaway Culprit identification significantly reduces manual effort in localizing inconsistencies with the DUT to particular parts of an NL specification and TL specification.

VI. Conclusion

Typical translation of unstructured NL to TL is *unstructured*, requiring users to manually inspect/correct complex TL outputs. Instead, SynthTL conducts *structured translation* of unstructured NL to TL, which enables users to exclusively validate simple TL sub-specifications and decompositions that mechanically compose to produce a TL output. Plus, structured translation enables LLMs, model checkers, and human users to meaningfully collaborate on an NL to TL translation task.

Acknowledgment

We thank Mohammad Rahmani Fadiheh, Haoze Wu, and the anonymous reviewers for their constructive comments and feedback. This work was supported in part by the National Science Foundation (NSF), under awards 2153936 and 2236855 (CAREER); the Defense Advanced Research Projects Agency (DARPA) under contract W912CG-23-C-0025 and subcontract from Galois, Inc.; and by the German Federal Ministry of Education and Research (BMBF), through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 16KIS1138).

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 2000.

[2] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, 1977.

[3] S. Vijayaraghavan and M. Ramanathan, "A practical guide for system verilog assertions," 2005.

[4] "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[5] H. Foster, "Part 1: The 2022 wilson research group functional verification study," Jan 2023.

[6] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," in *Computer Aided Verification* (C. Enea and A. Lal, eds.), (Cham), pp. 383–396, Springer Nature Switzerland, 2023.

[7] I. Gavran, E. Darulova, and R. Majumdar, "Interactive synthesis of temporal specifications from examples and natural language," *Proc. ACM Program. Lang.*, vol. 4, nov 2020.

[8] F. Fuggitti and T. Chakraborti, "Nl2ltl - a python package for converting natural language (nl) instructions to linear temporal logic (ltl) formulas," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23, AAAI Press, 2023.

[9] Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "NL2TL: Transforming natural languages to temporal logics using large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Dec. 2023.

[10] J. He, E. Bartocci, D. Ničković, H. Isakovic, and R. Grosu, "Deepstl: from english requirements to signal temporal logic," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 610–622, Association for Computing Machinery, 2022.

[11] C. Hahn, F. Schmitt, J. J. Tillman, N. Metzger, J. Siber, and B. Finkbeiner, "Formal specifications from natural language," 2022.

[12] J. X. Liu, Z. Yang, B. Schornstein, S. Liang, I. Idrees, S. Tellex, and A. Shah, "Lang2LTL: Translating natural language commands to temporal specification with large language models," in *Workshop on Language and Robotics at CoRL 2022*, 2022.

[13] C. Wang, C. Ross, Y.-L. Kuo, B. Katz, and A. Barbu, "Learning a natural-language to ltl executable semantic parser for grounded robotics," in *Proceedings of the 2020 Conference on Robot Learning* (J. Kober, F. Ramos, and C. Tomlin, eds.), vol. 155 of *Proceedings of Machine Learning Research*, pp. 1706–1718, PMLR, 16–18 Nov 2021.

[14] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "Gpt-4 technical report," 2024.

[15] Arm Ltd., *AMBA AHB Protocol Specification*, 2021.

[16] Y. Godhal, K. Chatterjee, and T. A. Henzinger, "Synthesis of amba ahb from formal specification: a case study," *International Journal on Software Tools for Technology Transfer*, vol. 15, pp. 585 – 601, 2011.

[17] E. Conrad, L. Titolo, D. Giannakopoulou, T. Pressburger, and A. Dutle, "A compositional proof framework for fretish requirements," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, (New York, NY, USA), p. 68–81, Association for Computing Machinery, 2022.

[18] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 372–381, 2005.

[19] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, (New York, NY, USA), p. 31–40, Association for Computing Machinery, 2008.

[20] A. Brunello, A. Montanari, and M. Reynolds, "Synthesis of LTL formulas from natural language texts: State of the art and research directions," in *26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain* (J. Gamper, S. Pinchinat, and G. Sciavicco, eds.), vol. 147 of *LIPIcs*, pp. 17:1–17:19, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[21] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[22] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[23] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko, "From Spot 2.0 to Spot 2.10: What's new?," in *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, vol. 13372 of *Lecture Notes in Computer Science*, pp. 174–187, Springer, Aug. 2022.

[24] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* (Y. Lakhnech and S. Yovine, eds.), (Berlin, Heidelberg), pp. 152–166, Springer Berlin Heidelberg, 2004.

[25] B. Walshe, "What is amba?," 2014.

[26] Arm Ltd., "Learn the architecture - an introduction to amba axi," 2022.

Recomposition: A New Technique for Efficient Compositional Verification

Ian Dardik
Carnegie Mellon University
Pittsburgh, PA, USA
idardik@andrew.cmu.edu

April Porter
University of Maryland, College Park
College Park, MD, USA
aporter3@terpmail.umd.edu

Eunsuk Kang
Carnegie Mellon University
Pittsburgh, PA, USA
eunsukk@andrew.cmu.edu

***Abstract*—Compositional verification algorithms are well-studied in the context of model checking. Properly selecting components for verification is important for efficiency, yet has received comparatively less attention. In this paper, we address this gap with a novel compositional verification framework that focuses on component selection as an explicit, first-class concept. The framework decomposes a system into components, which we then *recompose* into new components for efficient verification. At the heart of our technique is the *recomposition map* that determines how recomposition is performed; the component selection problem thus reduces to finding a good recomposition map. However, the space of possible recomposition maps can be large. We therefore propose heuristics to find a small portfolio of recomposition maps, which we then run in parallel. We have implemented our techniques in a model checker for the TLA⁺ language. In our experiments, we show that our tool achieves competitive performance with TLC–a well-known model checker for TLA⁺–on a benchmark suite of distributed protocols.**

I. Introduction

Model checking is an important tool for software, protocol, and algorithm development. *Compositional verification* is a paradigm in which a system is decomposed into components, which are then verified using a divide-and-conquer algorithm. To help model checking scale to large programs and specifications, compositional verification remains an important type of technique for combating the state explosion problem [1].

Most research papers on compositional verification assume that the components are pre-determined and focus solely on verification algorithms [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. However, *component selection*–that is, determining the set of decomposed components and the order in which they are verified–can greatly impact performance, in terms of both run time and state space size. Yet there are comparatively fewer model checking frameworks that investigate component selection, e.g. by automating decomposition [17], [18], [19]. Unfortunately, research into automated decomposition has seen limited success thus far; as Cobleigh et al. lament, decomposing a system is tough [17].

In this paper, we propose a new safety verification approach for symbolic specifications that is centered around component selection. In our approach, we begin by decomposing a system S into components $C_1, \dots, C_n$. Traditionally, a compositional verification algorithm is applied to these components to verify a system level property P, as shown in Fig. 1a. However, verifying these components may be less efficient than verifying the entire (monolithic) system directly without compositional techniques. Our key insight that addresses this shortcoming is to *recompose* the components into new components $D_1, \dots, D_m$ that we verify instead. For example, Fig. 1b shows D_1 composed of C_1 and C_3 while D_2 is composed of C_2.

The choice of how to recompose is determined by a *recomposition map* that maps C_i's to D_j's. Recomposition maps make component selection an explicit, first-class concept and lie at the heart of our technique. We will show that, in practice, there often exists a recomposition map that results in a compositional verification problem that is more efficient than verifying the monolithic specification directly.

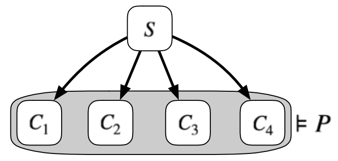
Additionally, we will show that our method is conducive to *specification reduction*. Specification reduction techniques, e.g. program slicing [20], [21], are generally considered separately from compositional verification. However, model checking with recomposition unites these two techniques under a single framework. For example, Fig. 1c shows a situation in which a partial recomposition map is used to reduce a specification with four-components to just the first three.

Ultimately, selecting components for efficient verification reduces to finding a suitable recomposition map. Therefore, we propose a technique for automatically selecting recomposition maps. We use heuristics to prune the large space of possible recomposition maps, which results in a small portfolio of maps that we run in parallel.

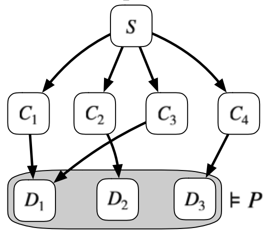
We have implemented our techniques in a model checker called "Recomp-Verify" for the TLA⁺ language [22]. In order to bring compositional verification to TLA⁺, we additionally propose a novel parallel composition operator for the language. We evaluate our techniques by comparing Recomp-Verify to TLC [23], a well-known model checker for TLA⁺. We show that recomposition can lead to large savings in terms of verification time and the size of the explored state space.

In summary, we make the following contributions: **(1) our main contribution, recomposition, which is a technique for efficient compositional verification,** (2) an automated method for finding efficient recomposition maps using parallelization and heuristics, (3) a definition for parallel composition for TLA⁺ specifications, and (4) a prototype model checker Recomp-Verify that implements our algorithm, along with an evaluation of Recomp-Verify against TLC on a benchmark of

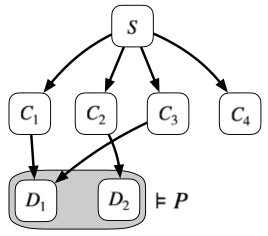
https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_18



(a) Traditional compositional verification.



(b) Compositional verification with recomposition.



(c) Specification reduction in the recomposition method.

Fig. 1: Comparing traditional compositional verification against our recomposition method.

```
──────────────────── MODULE TwoPhase ────────────────────
1   CONSTANT RMs
2   VARIABLES msgs, rmState, tmState, tmPrepared
3   vars ≜ ⟨msgs, rmState, tmState, tmPrepared⟩

4   Init ≜
5      ∧ msgs = {}
6      ∧ rmState = [rm ∈ RMs ↦ "working"]
7      ∧ tmState = "init"
8      ∧ tmPrepared = {}

9   RcvPrepare(rm) ≜
10     ∧ [type ↦ "Prepared",  theRM ↦ rm] ∈ msgs
11     ∧ tmState = "init"
12     ∧ tmPrepared′ = tmPrepared ∪ {rm}
13     ∧ UNCHANGED ⟨msgs, tmState, rmState⟩

14  SndPrepare(rm) ≜
15     ∧ rmState[rm] = "working"
16     ∧ msgs′ = msgs ∪ {[type ↦ "Prepared",  theRM ↦ rm]}
17     ∧ rmState′ = [rmState EXCEPT ![rm] = "prepared"]
18     ∧ UNCHANGED ⟨tmState, tmPrepared⟩

19  Next ≜
20        ∃ rm ∈ RMs :
21             ∨ SndPrepare(rm)
22             ∨ RcvPrepare(rm)
23                   ⋮
24  Spec ≜ Init ∧ □[Next]_vars
```

Fig. 2: A monolithic encoding of the Two Phase Commit Protocol.

distributed protocols.

II. Motivating Example

In this section, we describe the Two Phase Commit Protocol [24] to motivate our work and serve as a running example throughout the paper.

1) Protocol Description: In the Two Phase Commit Protocol, a *transaction manager* (TM) attempts to commit a transaction onto a pool of *resource managers* (RMs) in two phases. In the first phase, each RM starts in the *working* state as it attempts to commit the transaction. Any RM that can commit a transaction sends a *prepared* message to the TM. In the second phase, if every RM is prepared, the TM will issue a *commit* message to each RM; otherwise the TM will issue an *abort* message. The protocol assumes that the network can reorder, but not lose, messages. The key safety property is for each RM to remain *consistent*; i.e. no two RMs should disagree as to whether a transaction was committed or aborted.

2) TLA$^+$ Encoding: In Fig. 2, we show only the first (prepare) phase of the $TwoPhase$ specification, which we will refer to as TP for brevity. TP is a *parameterized* protocol, meaning that the set of RMs in the protocol is given as input. In the TP specification, the parameter is indicated on line 1 using the keyword CONSTANT.

TP defines a symbolic transition system (STS) over four state variables, which are declared on line 2 in Fig. 2. The variable $rmState$ is the state of each RM, the variables $tmState$ and $tmPrepared$ hold the state of the TM, and $msgs$ is the set of messages each machine sends over the network. Line 24 formally declares the STS with initial predicate $Init$ and transition relation $Next$. We show two actions, $SndPrepare$ and $RcvPrepare$, on lines 14 and 9 respectively. In TLA$^+$, actions are typically conjunctions of guards that specify when an action is enabled (lines 10-11 and 15) as well as primed variable expressions that specify transitions (lines 12 and 16-17). The UNCHANGED keyword on lines 13 and 18 indicate the frame conditions.

The key safety property for the Two Phase Commit Protocol is the invariant $Consistent$. We can encode this invariant as the following TLA$^+$ formula:

$$\forall\, rm1,\, rm2 \in RMs : \\ \neg(rmState[rm1] = \text{“aborted”} \wedge rmState[rm2] = \text{“committed”})$$

3) Model Checking TwoPhase: The TLC model checker can prove that a given finite instance of TP satisfies the property $Consistent$. A finite instance of a protocol substitutes a finite value for each parameter, e.g. a finite set of resource managers for RMs in TP. TLC performs explicit state model checking, meaning that it enumerates every possible state in the transition system. For nine resource managers, TLC is able to prove TP is safe after generating over 10 million states in nearly ten minutes. However, for ten resource managers, TLC fails to terminate in an hour after checking over 48 million states. In the following section, we will show how our approach can scale model checking TP to ten resource

```
──── MODULE RM ────
VARIABLES rmState
Init ≜
   ∧ rmState =
      [rm ∈ RMs ↦ "working"]
SndPrepare(rm) ≜
   ∧ rmState[rm] = "working"
   ∧ rmState' =
      [rmState EXCEPT ![rm] =
       "prepared"]
───────────────────
```

```
──── MODULE Env ────
VARIABLES msgs
Init ≜ msgs = {}
SndPrepare(rm) ≜
   ∧ msgs' = msgs ∪ {[type ↦
      "Prepared", theRM ↦ rm]}
RcvPrepare(rm) ≜
   ∧ [type ↦ "Prepared",
      theRM ↦ rm] ∈ msgs
   ∧ UNCHANGED ⟨msgs⟩
───────────────────
```

```
──── MODULE TM_1 ────
VARIABLES tmState
Init ≜ tmState = "init"
RcvPrepare(rm) ≜
   ∧ tmState = "init"
   ∧ UNCHANGED ⟨tmState⟩
───────────────────
```

```
──── MODULE TM_2 ────
VARIABLES tmPrepared
Init ≜ tmPrepared = {}
RcvPrepare(rm) ≜
   ∧ tmPrepared' =
      tmPrepared ∪ {rm}
───────────────────
```

Fig. 3: A decomposition of TP. Standard operators such as $Spec$, $Next$, $vars$, etc. are omitted for brevity.

managers.

4) Compositional Verification and Recomposition: Consider the specifications RM, Env, TM_1, and TM_2 shown in Fig. 3. These specifications represent a decomposition of TP; that is, TP is semantically equal to the parallel composition of the four specifications. We can generate a labeled transition system (LTS) for each of the four specifications in Fig. 3 and then use compositional verification techniques to answer the original model checking problem. For ten resource managers, this strategy enumerates a maximum of 261,002 states and terminates in 1 minute and 32 seconds.

Compositionally verifying the components above is more efficient than TLC, but *we can use recomposition to do even better*. Later, in example Ex. 2, we use recomposition to identify new components that are *optimal* in terms of minimum run time for verification. In general, recomposition can provide large savings in terms of run time and state space. In Sec. VII, we show experimentally that recomposition can reduce a model checking problem by *millions* of states.

III. Preliminaries

In this section we formally introduce *labeled transition systems (LTSs)*, the TLA+ language, and the compositional verification technique that we consider in this paper. Throughout this paper, we will use calligraphic font when referring to LTS variables (e.g. $\mathcal{D}$) and normal font when referring to TLA+ specifications (e.g. S).

A. Labeled Transition Systems

A labeled transition system (LTS) $\mathcal{D}$ is a tuple $(Q, \alpha\mathcal{D}, \delta, I)$ where Q is the set of states, $\alpha\mathcal{D}$ is the alphabet of $\mathcal{D}$, $\delta \subseteq Q \times \alpha\mathcal{D} \times Q$ is the transition relation, and I is a set of initial states. $\alpha\mathcal{D}$ must be a subset of $\mathbb{A}$, where $\mathbb{A}$ is the universe of all possible actions across all possible LTSs. We let $Reach(\mathcal{D})$ be the set of reachable states in $\mathcal{D}$. We define the parallel composition ($\|$) over LTSs in the usual way by synchronizing on actions common to both alphabets and interleaving on all other actions [25].

$$\begin{array}{llll} spec ::= Spec \triangleq Init \wedge \Box[Next]_{vars} & & conj ::= & \wedge\, expr \mid \wedge\, expr\ conj \\ init ::= Init \triangleq conj & & & \\ next ::= Next \triangleq \exists\, x \in D : disj & & disj ::= & \vee\, expr \mid \vee\, expr\ disj \\ expr ::= \text{arbitrary TLA+ expression} & & op ::= & id(p) \triangleq conj \end{array}$$

Fig. 4: Restricted TLA+ grammar for this paper.

We define an action-based behavior σ as an infinite sequence of actions, i.e. $\sigma \in \mathbb{A}^\omega$, and we let σ_i denote the ith action in σ. We denote the action-based semantics of an LTS $\mathcal{D}$ as a set of action-based behaviors $[\![\mathcal{D}]\!]_\alpha \subseteq \mathbb{A}^\omega$. It is the case that $\sigma \in [\![\mathcal{D}]\!]_\alpha$ if and only if there exists a sequence of states $q_0, q_1, \ldots \in Q^\omega$ such that $q_0 \in I$ and, for each nonnegative index i, either (1) $\sigma_i \in \alpha\mathcal{D}$ and $(q_i, \sigma_i, q_{i+1}) \in \delta$, or (2) $\sigma_i \notin \alpha\mathcal{D}$ and $q_i = q_{i+1}$. Condition (2) allows for *stuttering*, a concept which we will introduce in Sec. III-B.

There are two methods for encoding a safety property as an LTS. The first method is creating an *error LTS* that includes an error state–which we refer to as the π state–that acts as a sink for any action that causes a safety violation. The second method is creating a *property LTS* whose language defines the safe behaviors; property LTSs must be deterministic and must not include a π state. Any error LTS can be converted to a property LTS using steps two and three for assumption generation (Sec. 3) in [26]. We define property satisfaction over property LTSs as follows: an LTS $\mathcal{D}$ satisfies a property LTS $\mathcal{P}$ ($\mathcal{D} \models \mathcal{P}$) exactly when $[\![\mathcal{D}]\!]_\alpha \subseteq [\![\mathcal{P}]\!]_\alpha$. Note that our LTS semantics (with stuttering) properly handles alphabet refinement, and therefore it is unnecessary to consider alphabet restriction [25] in our definition of property satisfaction.

B. TLA+

In this paper we will refer to a TLA+ specification S as a syntactic entity that consists of constants, variables and operator definitions, etc. in the format shown in Fig. 2.

The initial state predicate, transition relation, and specification declaration are named $Init$, $Next$, and $Spec$ respectively. In this paper, $Init$, $Next$, and $Spec$ are restricted to the syntax of $init$, $next$, and $spec$ given by the grammar in Fig. 4. In $next$, the domain D does not contain state variables. We also restrict action definitions to the syntax of op, and no actions are referenced in the body of another action. In the grammar, $\Box$ is the *always* temporal operator. Expression $[Next]_{vars}$ is equal to $Next \vee (vars' = vars)$ and allows for *stuttering* states, i.e. consecutive states whose variables in $vars$ do not change.

We define several operators over a TLA+ specification S. The scoping operator ! references definitions in S, e.g. $TP!SndPrepare$ refers to the $SndPrepare$ action of TP in Fig. 2. The operators $\hat{\alpha}$ and α denote *symbolic actions* and *concrete actions* respectively. Symbolic actions are the action names in a specification, while concrete actions are the actions that may occur in a finite instance. For example, let TP_1 be the finite instance of TP with $RM = \{\text{"}rm1\text{"}\}$, then $\hat{\alpha}TP_1 = \hat{\alpha}TP = \{SndPrepare, RcvPrepare\}$ and $\alpha TP_1 = \{SndPrepare(\text{"}rm1\text{"}), RcvPrepare(\text{"}rm1\text{"})\}$. Additionally, we let βS denote the set of state variables in a specification or an expression, e.g. $\beta TP = \{msgs, rmState, tmState, tmPrepared\}$. For an operator $* \in \{\hat{\alpha}, \alpha, \beta\}$ and a set of specifications Z, the notation $*Z$ is short-hand for the union of $*z$, for each specification $z \in Z$.

To define the semantics of a TLA$^+$ formula, we first define a state as an assignment to all state variables. Then, the semantics of a TLA$^+$ formula is a set of behaviors, where a behavior is an infinite sequence of states. We indicate state-based semantics of a TLA$^+$ formula F as $[\![F]\!]_\beta$, the set of behaviors that satisfy F. For a TLA$^+$ specification S, we will often abbreviate $[\![S!Spec]\!]_\beta$ to simply $[\![S]\!]_\beta$. Given a TLA$^+$ property P, we say S satisfies P ($S \models P$) exactly when $[\![S]\!]_\beta \subseteq [\![P]\!]_\beta$.

We define the operator $\textsc{lts}(S)$, which converts a TLA$^+$ specification S into an LTS $\mathcal{D}$. $\textsc{lts}(S)$ can be realized by generating the full state graph for S and then labeling its edges with the concrete actions αS such that $\alpha\mathcal{D} = \alpha S$. Additionally, we define two operators for converting TLA$^+$ properties to an LTS. The first operator, $\textsc{err}(S, P)$, constructs an error LTS for S where violations of P lead to a π state. The second operator, $\textsc{prop}(S, P)$, builds a property LTS for S where no violation of P is possible. $\textsc{prop}(S, P)$ can be constructed from $\textsc{err}(S, P)$, as pointed out in Sec. III-A.

C. CRA-Style Compositional Verification

In this paper, we consider a style of compositional verification called *compositional reachability analysis* (CRA) [4], [8], [27]. Our recomposition framework requires a compositional verification algorithm that works for multiple components, and CRA-style techniques have reported success for verifying safety properties of multi-component systems [6].

CRA is used to check safety by composing the LTS for each component together in a hierarchical fashion; safety is proved if and only if the π state is unreachable in the overall system. Such algorithms generally derive their divide-and-conquer efficiency from two optimizations: intermediate minimization and short-circuiting. The former involves minimizing the state space of the intermediate LTSs with respect to observational equivalence [28] during composition. The latter optimization, short-circuiting, occurs when a strict subset of components are needed for verification to succeed. In this case, the remaining components (outside the strict subset) can be skipped, and hence short-circuiting provides a *dynamic* form of specification reduction.

IV. Parallel Composition in TLA$^+$

In this section, we introduce a new parallel composition operator over TLA$^+$ specifications. The operator is central to our recomposition algorithm and will allow us to define concepts such as decomposition and recomposition in Sec. V. The new operator is syntactic; in other words, the definition is entirely in terms of TLA$^+$ syntax, and does not involve explicitly enumerating the state space. To avoid confusion between the parallel composition operator $\|$ over LTSs (Sec. III-A), we will denote the TLA$^+$ parallel composition operator using $/\!/$. We will use the notation $/\!/Z$ to denote the composition over a set of specifications Z. We now define $/\!/$ in the usual way, by synchronizing common actions between specifications and interleaving all others actions [25].

```
──────────── MODULE T₁ ────────────
VARIABLES
   msgs, tmState, tmPrepared
Init ≜
   ∧ msgs = {}
   ∧ tmState = "init"
   ∧ tmPrepared = {}
SndPrepare(rm) ≜
   ∧ msgs′ = msgs ∪
      {[type ↦ "Prepared",
        theRM ↦ rm]}
   ∧ UNCHANGED
      ⟨tmState, tmPrepared⟩
RcvPrepare(rm) ≜
   ∧ [type ↦ "Prepared",
      theRM ↦ rm] ∈ msgs
   ∧ tmState = "init"
   ∧ tmPrepared′ =
      tmPrepared ∪ {rm}
   ∧ UNCHANGED
      ⟨msgs, tmState⟩
───────────────────────────────────
```

```
──────────── MODULE T₂ ────────────
VARIABLES
   tmState, tmPrepared
Init ≜
   ∧ tmState = "init"
   ∧ tmPrepared = {}
RcvPrepare(rm) ≜
   ∧ tmState = "init"
   ∧ tmPrepared′ =
      tmPrepared ∪ {rm}
   ∧ UNCHANGED ⟨tmState⟩
───────────────────────────────────
```

Fig. 5: Intermediate decomposed specifications T_1 and T_2 in the TP example.

Definition 1 (Parallel Composition)**.** Let S and T be TLA$^+$ specifications with distinct state variables. We define $S /\!/ T$ as follows. First, $S /\!/ T$ contains exactly the constants and state variables in S and T. Second, in $S /\!/ T$, we define $vars \triangleq (S!vars) \circ (T!vars)$, where $\circ$ is the sequence concatenation operator. Finally, in $S /\!/ T$, we define $Spec \triangleq Init \wedge [Next]_{vars}$ where $Init \triangleq S!\text{Init} \wedge T!\text{Init}$ and $Next$ is defined as follows:

$$\bigvee_{A \,\in\, \hat{\alpha}S \cup \hat{\alpha}T} \begin{cases} \exists d \in D \ : \ S!A(d) \ \wedge \ T!A(d) \\ \quad \text{if } A \in \hat{\alpha}S \text{ and } A \in \hat{\alpha}T \\ \exists d \in D \ : \ S!A(d) \ \wedge \ T!vars' = T!vars \\ \quad \text{if } A \in \hat{\alpha}S \text{ and } A \notin \hat{\alpha}T \\ \exists d \in D \ : \ T!A(d) \ \wedge \ S!vars' = S!vars \\ \quad \text{if } A \notin \hat{\alpha}S \text{ and } A \in \hat{\alpha}T \end{cases}$$

Notice that Def. 1 defines parallel composition in terms of TLA$^+$ syntax, and hence does not increase the expressivity of the language. While the operator itself is novel, this technique is briefly discussed by Lamport [22].

Example 1. By Def. 1, $TP = RM /\!/ Env /\!/ TM_1 /\!/ TM_2$. Furthermore, consider specifications T_1 and T_2 from Fig. 5. Notice that $TP = RM /\!/ T_1$, $T_1 = Env /\!/ T_2$, and $T_2 = TM_1 /\!/ TM_2$.

The following theorem shows that parallel composition behaves exactly as we expect if we convert a TLA$^+$ specification to an LTS. We prove this theorem using more general semantics for TLA+ specifications, namely action-state-based semantics. We include a proof in Appendix A of our technical report [29].

Theorem 1. $[\![\textsc{lts}(S \mathbin{/\!/} T)]\!]_\alpha = [\![\textsc{lts}(S) \parallel \textsc{lts}(T)]\!]_\alpha$.

V. Model Checking with Recomposition

In this section, we propose our algorithm for verifying symbolic specifications. We begin by introducing the algorithm in Sec. V-A. Subsequently, we provide details for decomposition (Sec. V-B), static specification reduction (Sec. V-C), and compositional verification (Sec. V-D). Finally, we conclude this section with a correctness analysis of the algorithm in Sec. V-E.

A. The Recomp-Verify Algorithm

1) Algorithm Overview: Our algorithm solves a model checking problem $S \models P$, where S and P are both written in TLA+. The algorithm begins by decomposing S into n components $C_1, \dots, C_n$, each of which is also a TLA+ specification. The decomposition algorithm ensures two key properties upon termination: (P1) $S = C_1 \mathbin{/\!/} \cdots \mathbin{/\!/} C_n$ and (P2) the first component, C_1, contains all state variables that occur syntactically in P. Property (P1) ensures soundness of the decomposition, while property (P2) allows us to build the safety property $\mathcal{P}$ described in the following paragraph.

After decomposition, the algorithm *recomposes* the C_i components into new components D_P and $D_1, \dots, D_m$. These new components define the following compositional verification problem that is equivalent to the original: $\textsc{lts}(D_1) \parallel \dots \parallel \textsc{lts}(D_m) \models \mathcal{P}$, where $\mathcal{P} = \textsc{prop}(D_P, P)$. For $\textsc{prop}(D_P, P)$ to be well-formed, D_P must contain every state variable that occurs in P. Therefore, we require D_P to be composed of (at least) C_1, as C_1 must contain every state variable that occurs in P by property (P2) of decomposition. We formally capture this requirement, as well as the choice of how to perform recomposition, in the following definition.

Definition 2 (Recomposition Map)**.** A *recomposition map* is a surjective function $f : \{C_1, \dots, C_n\} \to \{d_P, d_1, \dots, d_m\}$ such that $f(C_1) = d_P$.

In Def. 2, the d_j's in the co-domain are intended as a placeholder for constructing each D_j. In particular, we will define each recomposed component as $D_j = \mathbin{/\!/\!/} f^{-1}(d_j)$, the parallel composition of one or more C_i components. Therefore, the restriction $f(C_1) = d_P$ implies that D_P will be composed of C_1 as intended. Finally, once each D_j is constructed, we solve the compositional verification problem.

2) Algorithm Details: We present our model checking algorithm in Alg. 1. The algorithm accepts several inputs, including a *recomposition strategy*. A recomposition strategy ρ is a function that maps C_i components to a pair (f, m), where f is a total recomposition map and m is the number of D_j components. In other words, the recomposition strategy determines which recomposition map is used. In the remainder of this section we assume ρ is given; we discuss recomposition strategy selection in Sec. VI.

Algorithm 1 Recomp-Verify

```
Input: Specification $S$, property $P$, recomposition strategy $\rho$
Output: If $S \models P$
1: $C_1, \dots, C_n = \textsc{Decompose}(S, P)$
2: $f, m \leftarrow \rho(C_1, \dots, C_n)$
3: $f, m \leftarrow \textsc{Static-Reduce}(f, m)$
4: $D_P \leftarrow \mathbin{/\!/\!/} f^{-1}(d_P)$            $\triangleright\ f^{-1}(d_P) \subseteq \{C_1, \dots, C_n\}$
5: for $j \in \{1, \dots, m\}$ do
6:     $D_j \leftarrow \mathbin{/\!/\!/} f^{-1}(d_j)$            $\triangleright\ f^{-1}(d_j) \subseteq \{C_2, \dots, C_n\}$
7: return $\textsc{Comp-Verify}(D_1, \dots, D_m, D_P, P)$
```

Alg. 1 begins by decomposing S into components on line 1. The strategy ρ selects a recomposition map on line 2, which is possibly statically reduced on line 3. We provide more detail for decomposition and static specification reduction in Sec. V-B and Sec. V-C respectively. Next, on lines 4-6, we perform recomposition using the recomposition map f. On line 4, we define D_P to be the parallel composition of each C_i component in the pre-image $f^{-1}(d_P)$. Similarly, on line 6, we define each D_j to be the parallel composition of each C_i component in the pre-image $f^{-1}(d_j)$. Finally, on line 7, we solve the compositional verification problem for the recomposed components (D_j's); we provide more detail for this step in Sec. V-D.

Example 2. In this example we analyze Alg. 1 given the input TP, $Consistent$, and a hand-crafted optimal recomposition strategy ρ_{opt}. Line 1 of Alg. 1 produces the components RM, Env, TM_1, and TM_2 from Fig. 3. On line 2, ρ_{opt} chooses $m = 2$ and f such that $f(RM) = d_P$, $f(Env) = f(TM_1) = d_1$, and $f(TM_2) = d_2$. Static specification reduction on line 3 has no effect on f and m. Recomposition (lines 4-6) reduces the original model checking problem to $\textsc{lts}(Env \mathbin{/\!/} TM_1) \parallel \textsc{lts}(TM_2) \models \textsc{prop}(RM, Consistent)$, which we solve on line 7. Whereas the example in Sec. II verifies four specifications (for RM, Env, TM_1, TM_2), this example verifies three (for RM, $Env \mathbin{/\!/} TM_1$, TM_2). The strategy ρ_{opt} in this example reduces the maximum state space by 1,027 states and improves the model checking time from 1 minute 32 seconds to 51 seconds.

B. Decomposition

In this section, we present an algorithm for decomposing a symbolic specification S into n components $C_1 \dots C_n$. Our algorithm guarantees the following two properties: (P1) $S = C_1 \mathbin{/\!/} \cdots \mathbin{/\!/} C_n$ and (P2) $\beta P \subseteq \beta C_1$. We provide a correctness argument for these two properties in Sec. V-E.

1) Decomposition Algorithm: Each step of the algorithm splits a specification T_i into two specifications C_{i+1} and T_{i+1} such that $T_i = C_{i+1} \mathbin{/\!/} T_{i+1}$. We note the following two corner cases: $T_0 = S$ and $C_n = T_{n-1}$. The algorithm splits a specification across two phases: *state variable partitioning* and *specification slicing*. The former partitions the state variables of T_i into two sets V_C and V_T, while the latter slices T_i into C_{i+1} and T_{i+1} that contain the variables V_C and V_T respectively. We present the algorithm in Alg. 2. We now explain state variable partitioning and specification slicing in detail across the following two sections.

Example 3. We explain Alg. 2 given TP and $Consistent$. The algorithm begins with the partition $V_C = \{rmState\}$ and $V_T = \{msgs, rmState, rmPrepared\}$ on line 1; we explain partitioning in Sec. V-B2. Next, on lines 6-7, the algorithm slices TP into RM (Fig. 3) and T_1 (Fig. 5). The state variables of T_1 are subsequently partitioned into $V_C = \{msgs\}$ and $V_T = \{rmState, rmPrepared\}$ on line 9. The algorithm continues in this fashion until $V_T = \emptyset$, i.e. no partition is possible. The algorithm will then exit the loop and return the components RM, Env, TM_1, TM_2 on line 12.

Algorithm 2 DECOMPOSE

```
Input: Specification $S$, Safety Property $P$
Output: $C_1, \dots, C_n$ with properties (P1) and (P2)
 1: $V_C, V_T \leftarrow \textsc{Partition}(S, \beta P)$
 2: if $V_T = \emptyset$ then
 3:     return $S$
 4: $T_0 \leftarrow S,\ i \leftarrow 0$
 5: while $V_T \neq \emptyset$ do
 6:     $C_{i+1} \leftarrow \textsc{Slice}(T_i, V_C)$
 7:     $T_{i+1} \leftarrow \textsc{Slice}(T_i, V_T)$
 8:     $v \in \beta T_{i+1}$   ▷ Nondeterministically choose a variable
 9:     $V_C, V_T \leftarrow \textsc{Partition}(T_{i+1}, \{v\})$
10:     $i \leftarrow i + 1$
11: $n \leftarrow i + 1,\ C_n \leftarrow T_i$
12: return $C_1, \dots, C_n$
13: procedure $\textsc{Partition}(T, V)$
14:     $V_C \leftarrow \textsc{Fix}(\textsc{Occurs}_T, V)$
15:     $V_T \leftarrow \beta S - V_C$
16:     return $V_C, V_T$
17: procedure $\textsc{Occurs}_S(V)$
18:     return $\bigcup_{A \in \hat{\alpha} S} \{\beta c \mid c \in \text{Conj}(A) \text{ and } \beta c \cap V \neq \emptyset\}$
19: procedure $\textsc{Fix}(op, X)$
20:     $Y \leftarrow X \cup op(X)$
21:     if X = Y then return $X$
22:     return $Y \cup \textsc{Fix}(op, Y)$
```

2) State Variable Partitioning: Given a specification T_i, the partitioning phase partitions the variables βT_i into two sets V_C and V_T. The partition procedure appears twice in Alg. 2. The first occurrence, on line 1, determines the state variables that will appear in C_1; therefore, to uphold property (P2), we partition on βP. In the second appearance, on line 9, we choose just *one* variable in attempt to produce as many components as possible (ideally, one component per state variable). We are free to choose the one variable nondeterministically because the order of decomposed components is inconsequential; this is due to the fact that the ordering is ultimately determined by a recomposition map in Alg. 1.

The partition procedure in Alg. 2 also guarantees that the state variables in each partition will constitute a well-formed slice according to the grammar in Fig. 4. For example, if a specification contains the expression $a = b + 1$, then a and b should be grouped together into the same partition. To accomplish this, we let V_C be V *plus* any variables that occur within the same expression, repeated until fix-point. More formally, we let $V_C = \textsc{Fix}(\textsc{Occurs}_S, V)$ (line 14), where $\textsc{Fix}$ invokes the $\textsc{Occurs}_S$ procedure, initially on V, until a fix-point is reached. Finally, we choose V_T to be the remainder of the state variables in T_i (line 15).

Example 4. Notice that $\beta Consistent = \{rmState\}$ and $\textsc{Fix}(\textsc{Occurs}_{TP}, \{rmState\}) = \{rmState\}$. Therefore, the first partition (line 1) will be $V_C = \{rmState\}$ and $V_T = \{msgs, tmState, tmPrepared\}$. In the second partition (lines 8-9), we arbitrarily choose $v = msgs$, which results in $V_C = \{msgs\}$ and $V_T = \{tmState, tmPrepared\}$.

3) Specification Slicing: The specification slicing phase restricts a specification T_i to a given subset of its variables V. Slicing can be seen as the inverse of parallel composition. For example, consider a system specification M with action $Action$ and state variables var_1 and var_2:

$$\begin{aligned} Action \triangleq\ & \land\, var_1{}' = \text{“val1”} \\ & \land\, var_2{}' = \text{“val2”} \end{aligned}$$

Given the variable partition $\{var_1\}$, $\{var_2\}$, we can view M as the composition of two components M_1 and M_2 that respectively define: $Action \triangleq var_1{}' = \text{“val1”}$ and $Action \triangleq var_2{}' = \text{“val2”}$. In particular, we have $M_1 = \textsc{Slice}(M, \{var_1\})$, $M_2 = \textsc{Slice}(M, \{var_2\})$, and $M = M_1 /\!/ M_2$. In the TP example, this corresponds to $TP = RM /\!/ T_1$ in Ex. 3. We include more details on slicing, including the definition for the slicing procedure, in Appendix B of our technical report [29].

C. Static Specification Reduction

In Sec. V-A, we require recomposition strategies to produce a total recomposition map. Total recomposition maps apply verification to every component; however, in some cases, not every component is necessary for verification. Therefore, in the following paragraph, we introduce a technique for statically detecting a subset of components that are necessary for verification. In Alg. 1, the procedure $\textsc{Static-Reduce}(f, m)$ on line 3 restricts the domain of f to this subset and reduces the codomain and m accordingly so f remains surjective.

The subset of necessary components is those whose alphabets may affect–either directly or indirectly–the actions of C_1, and therefore may prevent the entire system from reaching an error. More formally, the subset of components is $\bigcup_i X_i$, where $X_0 = \{C_1\}$ and $X_{i+1} = \{C_j \mid \hat{\alpha} C_j \cap \hat{\alpha} X_i \neq \emptyset\}$. In Appendix C of our technical report [29], we show that it is only necessary to consider the first $n + 1$ terms–where n is the number of C_i components–when computing the union of the X_i's. Intuitively, X_1 is the set of components that may directly prevent C_1 from reaching an error, while X_2, X_3, etc. may indirectly prevent an error.

Example 5. We now introduce $\mathit{TPCounter}$, an extension to TP. $\mathit{TPCounter}$ is identical to TP, except it includes one more state variable $\mathit{counter}$ and one more action $\mathit{Increment}$. In the initial state, $\mathit{counter}$ is equal to zero. Each original action from TP leaves $\mathit{counter}$ unchanged, while $\mathit{Increment}$ increments $\mathit{counter}$ by one and leaves all other state variables unchanged. The $\mathit{Increment}$ action is always enabled, and therefore $\mathit{TPCounter}$ is an infinite-state protocol.

Consider model checking $\mathit{TPCounter} \models \mathit{Consistent}$ with Alg. 1. Decomposition (line 1) produces five components: RM, Env, TM_1, TM_2, and $\mathit{Counter}$, where $\mathit{Counter}$ has one state variable $\mathit{counter}$ and one action $\mathit{Increment}$. $\mathit{Counter}$ is the only specification with the action $\mathit{Increment}$ and, therefore, does not synchronize with the actions in the other four specifications. Therefore, $\mathit{Counter}$ cannot affect the safety of C_1. Formally, $X_0 = \{\mathit{RM}\}$, $X_1 = \{\mathit{RM}, \mathit{Env}\}$, $X_2 = \{\mathit{RM}, \mathit{Env}, \mathit{TM}_1, \mathit{TM}_2\}$, $X_3 = X_2$, etc. so $\mathit{Counter}$ is not a necessary component. STATIC-REDUCE will therefore omit $\mathit{Counter}$ from the domain of any given recomposition map, causing Alg. 1 to successfully terminate.

D. Compositional Verification

We present a CRA-style compositional verification algorithm in Alg. 3. The algorithm works by iteratively composing the LTS for each component D_j together (line 5) until the π state becomes unreachable, in which case verification succeeds (lines 3 and 7). If the π state remains reachable by the end of the algorithm, however, then we report a failure (line 8). The algorithm performs intermediate minimization on lines 1 and 5. In general, there are many options for which components–or composition of components–to minimize [12]. We choose to only minimize components because we observed that minimizing the composition of components was generally slow. In essence, this algorithm is an abstraction-refinement loop where each new component lowers the abstraction by introducing more state variables.

Example 6. Consider TP with ten resource managers and the optimal mapping f from Ex. 2, where $D_P = \mathit{RM}$, $D_1 = \mathit{Env} /\!/ \mathit{TM}_1$, and $D_2 = \mathit{TM}_2$. Line 1 of Alg. 3 will generate an LTS for D_P with 477,454 states, including a π state. Minimization reduces D_P to 13,291 states. Due to a reachable π state, the algorithm proceeds into the loop on line 4. Next, on line 5, the algorithm generates an LTS for D_1 with 3,072 states, which reduces to 1,026 states after minimization. Composing this LTS with $\mathcal{D}$ (line 5) retains the π state (line 6) so we loop again. The algorithm continues in this fashion until a π state is no longer reachable, and we return a positive answer (line 6 and 7). A maximum of 481,550 states are needed in memory at once.

E. Correctness Analysis

In this section, we show that Alg. 1 is sound but not complete. To establish this result, we first provide lemmas for the correctness of decomposition (Lem. 1), static specification reduction (Lem. 2), and compositional verification (Lem. 3). Next, we show that reducing the monolithic model checking problem to compositional verification is correct (Lem. 4). Finally, we present Thm. 2 that shows that Alg. 1 is sound.

Algorithm 3 COMP-VERIFY

```
Input: $D_1, \dots, D_m, D_P, P$
Output: If $\text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_m) \models \mathcal{P}$
1: $\mathcal{D} \leftarrow \mathit{Min}(\text{ERR}(D_P, P))$        ▷ $\mathcal{P} = \text{ERR}(D_P, P)$
2: if $\pi \notin \mathit{Reach}(\mathcal{D})$ then
3:     return $true$
4: for $j \in \{1, \dots, m\}$ do
5:     $\mathcal{D} \leftarrow \mathcal{D} \parallel \mathit{Min}(\text{LTS}(D_j))$
6:     if $\pi \notin \mathit{Reach}(\mathcal{D})$ then
7:         return $true$
8: return $false$
```

Lemma 1. *Algorithm 2 ensures (P1) $S = C_1 /\!/ \cdots /\!/ C_n$ and (P2) $\beta C_1 \subseteq \beta P$ upon termination.*

Proof. Sketch. We prove property (P1) by establishing the following loop invariant in Alg. 2 on line 5: $S = C_1 /\!/ \cdots /\!/ C_i /\!/ T_i$. The proof for property (P2) follows in two steps. First, $\beta P \subseteq V_C$ because V_C (in the first partition) is defined by a monotonically increasing operation (FIX) on βP. Second, $\beta P \subseteq \beta C_1$ because $C_1 = \text{SLICE}(S, V_C)$ will contain exactly the state variables in V_C. □

Lemma 2. *$S \models P$ if and only if $/\!/(\bigcup_i X_i) \models P$.*

Proof. We prove this theorem in Appendix C of our technical report [29]. □

Lemma 3. *$\text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_m) \models \mathcal{P}$ if and only if there exists a $k \in \{0 \dots m\}$ such that $\pi \notin \mathit{Reach}(\text{ERR}(D_P, P) \parallel \text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_k))$.*

Proof. Sketch. The forwards case ($\Rightarrow$) follows by choosing $k = m$. For the backwards case ($\Leftarrow$), we assume that k is given such that $0 \leq k \leq m$ and $\pi \notin \mathit{Reach}(\text{ERR}(D_P, P) \parallel \text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_k))$. Notice that, by construction, none of $\text{LTS}(D_{k+1}), \dots, \text{LTS}(D_m)$ contains a π state, and therefore neither will $\mathit{Reach}(\text{ERR}(D_P, P) \parallel \text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_m))$. □

Lemma 4. *$S \models P \iff \text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_m) \models \mathcal{P}$.*

Proof.

$$
\begin{aligned}
& S \models P && (1)\\
\iff & \pi \in \mathit{Reach}(\text{ERR}(S, P)) && (2)\\
\iff & \pi \in \mathit{Reach}(\text{ERR}(C_1 /\!/ \cdots /\!/ C_n, P)) && (3)\\
\iff & \pi \in \mathit{Reach}(\text{ERR}(D_P /\!/ D_1 /\!/ \cdots /\!/ D_m, P)) && (4)\\
\iff & \pi \in \mathit{Reach}(\text{ERR}(D_P, P) \parallel \dots \parallel \text{ERR}(D_m, P)) && (5)\\
\iff & \pi \in \mathit{Reach}(\text{ERR}(D_P, P) \parallel \dots \parallel \text{LTS}(D_m)) && (6)\\
\iff & \text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_m) \models \text{PROP}(D_P, P) && (7)
\end{aligned}
$$

Biconditional (2) holds because P is a safety property, (3) by Lem. 1 property (P1), (4) by the definition for each D_j (and Lem. 2 in the case that f is partial), (5) by Thm. 1, (6) by Lem. 1 property (P2) and Def. 2 ($f(C_1) = d_P$), and (7) because P is a safety property. Finally, the theorem follows because $\mathcal{P} = \textsc{prop}(D_P, P)$. □

Theorem 2. *If Alg 1 terminates, then it returns true (model checking succeeds) if and only if $S \models P$.*

Proof. The result follows from Lem. 3 and Lem. 4. □

While Thm. 2 shows that Alg. 1 is sound, the algorithm is not complete, even if we limit S to be a finite-state specification. This is because a given component D_j may be infinite-state, in which case LTS construction will fail to terminate on line 1 or 5 in Alg. 3. We address this limitation in Sec. VI-B by using a portfolio of strategies that includes the *monolithic strategy*.

VI. Choosing Efficient Recomposition Maps

In this section, we address the problem of designing recomposition strategies, i.e. choosing efficient recomposition maps. Rather than finding a single recomposition strategy, we propose running Alg. 1 with a portfolio of strategies in parallel. The primary challenge is determining which strategies to use, since the number of possible recomposition maps grows large as the number of components increases. We therefore propose a heuristic for pruning the search space of recomposition maps in Sec. VI-A. We then choose a small portfolio of strategies based on this heuristic in Sec. VI-B.

A. Recomposition Map Reduction Heuristic

Any heuristic for pruning the search space of recomposition maps should be tailored to the compositional verification algorithm being used. Since we use a CRA-style verification algorithm, we design our heuristic to find component orderings that can take advantage of short-circuiting. In particular, the heuristic identifies recomposition maps that order D_j components that are least likely to be necessary for verification *last*.

Our heuristic is to choose recomposition maps that respect the *data flow* partial order $\preccurlyeq$ over the C_i components. This is a novel partial order that attempts to find dynamic specification reduction–i.e. short-circuiting–by refining our static specification reduction scheme. Intuitively, the partial order will order the components based on how far removed their state variables are from impacting verification.

More formally, we compute the data flow partial order based on the indexed sets X_i introduced in Sec. V-C. These sets cumulatively capture the components that may interact–either directly or indirectly–with the actions of C_1. First, we build new indexed sets E_i defined as $E_0 = X_0$ and $E_{i+1} = X_{i+1} \setminus X_i$. While the X_i's are cumulative, each E_i captures only the additional components in each X_i. Intuitively, the components in E_i are i steps removed from affecting the variables in C_1, and hence i steps removed from impacting verification (by property (P2) of decomposition).

Second, we build indexed sets F_i that capture the *data flow* from each component in E_i to the components in E_{i+1}. We define $F_0 = \emptyset$ and:

$$F_{i+1} = \left\{ (C_j, C_k) \;\middle|\; \begin{array}{c} C_j \in E_i \text{ and } C_k \in E_{i+1} \\ \text{and } \hat{\alpha} C_j \cap \hat{\alpha} C_k \neq \emptyset \end{array} \right\}$$

Finally, let $F = \bigcup_i F_i$; we define the data flow partial order $\preccurlyeq$ to be the reflexive transitive closure of F. In Appendix D of our technical report [29], we show formally that the partial order refines the static specification reduction scheme from Sec. V-C.

Example 7. For TP and $\mathit{Consistent}$, $E_0 = \{\mathit{RM}\}$, $E_1 = \{\mathit{Env}\}$, and $E_2 = \{\mathit{TM}_1, \mathit{TM}_2\}$. Moreover, $F = \{(\mathit{RM}, \mathit{Env}), (\mathit{Env}, \mathit{TM}_1), (\mathit{Env}, \mathit{TM}_2)\}$ and the data flow partial order is the reflexive transitive closure of this set. Intuitively, the partial order shows that TM_1 and TM_2 can only affect the variables in RM–i.e. the variables $\beta\,\mathit{Consistent}$ needed for verification–*indirectly* by interacting with the Env component.

To further reduce the search space of maps, we extend the data flow partial order to a total order $\leqslant$, i.e. $\preccurlyeq \subseteq \leqslant$. We build the total order by breaking ties between incomparable components C_i and C_j by requiring $C_i \leqslant C_j$ if and only if C_i's state variables have fewer syntactic appearances than C_j's in the original specification S. In the case that the variables of C_i and C_j have the same number of appearances in S, we break the tie arbitrarily.

B. Choosing a Portfolio of Strategies

In this section, we describe four recomposition strategies that comprise our portfolio. For simplicity, we describe the strategies assuming that the components $C_1, \ldots, C_n$ have been reordered according to the total order $\leqslant$ described in Sec. VI-A. The four strategies are (S1) the identity strategy, in which $m = n - 1$ and $f(C_i) = d_i$ for all i, (S2) a "bottom heavy" strategy in which we choose $m = 1$ and f such that $f(C_1) = d_P$ and $f(C_i) = d_1$ for all $i > 1$, (S3) a "top heavy" strategy in which we choose $m = 1$ and f such that $f(C_n) = d_1$ and $f(C_i) = d_P$ for all $i < n$, and (S4) the *monolithic* strategy, where $m = 0$ and $f(C_i) = d_P$ for all i.

Example 8. In TP, the state variables of TM_2 occur fewer times than the variables of TM_1. Therefore, the total ordering from Sec. VI-A is: $\mathit{RM}, \mathit{Env}, \mathit{TM}_2, \mathit{TM}_1$. Then, for each strategy, we have: (S1) $m = 3$, $f(\mathit{RM}) = d_P$, $f(\mathit{Env}) = d_1$, $f(\mathit{TM}_2) = d_2$, and $f(\mathit{TM}_1) = d_3$; (S2) $m = 1$, $f(\mathit{RM}) = d_P$, and $f(\mathit{Env}) = f(\mathit{TM}_2) = f(\mathit{TM}_1) = d_1$; (S3) $m = 1$, $f(\mathit{RM}) = f(\mathit{Env}) = f(\mathit{TM}_2) = d_P$, and $f(\mathit{TM}_1) = d_1$; and (S4) $m = 0$ and $f(\mathit{RM}) = f(\mathit{Env}) = f(\mathit{TM}_2) = f(\mathit{TM}_1) = d_P$.

As a note regarding the correctness analysis from Sec. V-E, including the monolithic strategy in the portfolio ensures termination. Therefore, our parallel approach is complete for finite state specifications S.

VII. Experimental Results

A. Implementation

We have created a model checker called Recomp-Verify that can verify safety for TLA+ specifications. The model checker is a prototype research tool that implements Alg. 1 in the Python, Java, and Kotlin programming languages. The model checker also supports running multiple instances of Alg. 1 in parallel, and returns the first result to finish. Our tool is available in a public repository [30].

B. Experiments

We evaluate Recomp-Verify against TLC on a benchmark of distributed protocols [31], plus the tla-twophase-counter protocol that we introduce in Ex. 5. Our evaluation is driven by two research questions. First, (**RQ1**) can hand-written recomposition maps provide more efficient verification than TLC? If this is the case, we then ask whether our technique is still performant when automating the search for recomposition maps. More precisely, (**RQ2**) is the performance of Recomp-Verify (using a parallel, portfolio strategy) competitive with TLC when each tool is allotted four threads?

In our experiments, we use TLC^1 and TLC^4 to respectively denote TLC run with one and four parallel threads; this is a built-in option for the tool. Recomp-Verify^1 is the version of our tool with one thread and hand-crafted maps, while Recomp-Verify^4 denotes the version that uses four threads to run the portfolio of recomposition strategies (S1-4) from Sec. VI in parallel. We report the fastest strategy for Recomp-Verify^4 in the "Strat." column in Fig. 6. Additionally, in the implementation of Recomp-Verify^4, we use TLC^1 for running the monolithic strategy (S4), since TLC is far more efficient than our research prototype for monolithic model checking. For example, in ex-quorum-leader-election-6 in Fig. 6, Recomp-Verify^1 uses an optimal single-threaded strategy, yet is slower than TLC^1–and therefore Recomp-Verify^4 too.

Every experiment in this paper was run on an Apple MacBook Pro with 32GB of memory and an M1 processor. For each benchmark, we report the total run time using the Unix *time* utility as well as the maximum number of states checked. We use TO to indicate a timeout after ten minutes and OM to indicate a program crash due to reaching the memory limit given a 25GB allotment. For TLC's maximum state count, we use the number of unique states that the tool reports. For Recomp-Verify, we use the maximum between (1) the number of unique states generated for each component and, for each iteration, (2) the number of states that results from composition in Alg. 3 on line 5. For Recomp-Verify^1, we also report the number of components that result from decomposition (n), the number of recomposed components (m), and the number of recomposed components that were checked (k).

C. Results and Discussion

1) RQ1: We show our results in Fig. 6. In terms of state space, TLC^1 enumerates at least as many states as Recomp-Verify^1 in every case. For six of the benchmarks that both tools verified, recomposition reduced the state size by *millions* of states. Moreover, Recomp-Verify^1 short-circuits ($k < m$) for eight benchmarks, each of which has a significantly smaller state space than TLC^1. Finally, Recomp-Verify^1–but not TLC^1–was able to verify the one infinite state benchmark (tla-twophase-counter) via static specification reduction.

In terms of verification speed, Recomp-Verify^1 and TLC^1 both outperform each other in fourteen benchmarks, and tie in five cases. However, Recomp-Verify^1 completes more benchmarks, verifying twenty-nine benchmarks while TLC^1 verifies twenty-four. Generally speaking, Recomp-Verify^1 is more performant on larger benchmarks; on benchmarks with over a million states, TLC^1 is faster in two cases while Recomp-Verify^1 is faster in at least six cases. We therefore answer RQ1 by concluding that hand-crafted maps *can* provide more efficiency than TLC.

2) RQ2: The results for TLC^4 and Recomp-Verify^4 are similar to the single threaded versions. In terms of state space, Recomp-Verify^4 always enumerates the same number of (or fewer) states than TLC^4, and also exhibits large savings in the millions for six benchmarks. We note that Recomp-Verify^4 short-circuited every time that Recomp-Verify^1 short-circuited, which showcases the effectiveness of the data flow heuristic.

In terms of verification speed, Recomp-Verify^4 is faster for eleven benchmarks, TLC^4 is faster for fourteen, and the tools tie for eight benchmarks. However, Recomp-Verify^4 verifies more benchmarks, completing thirty-two, while TLC^4 completes twenty-four. While threading made TLC faster for the smaller benchmarks, it did not help the tool verify more benchmarks. On the other hand, threading helped Recomp-Verify to verify three more benchmarks. Generally speaking, Recomp-Verify^4 outperforms TLC^4 for large benchmarks and is competitive with TLC^4 for smaller ones, and therefore we answer RQ2 in the affirmative.

3) Discussion: In Fig. 6, the Strat. column shows that the winning strategy for Recomp-Verify^4 varies depending on the given benchmark. This observation suggests that using a portfolio of strategies may be necessary for efficient verification. Notably, among the benchmark problems, we found that the bottom heavy strategy (S2) did not perform well. Most likely, this is because the second component in this strategy is too large, and therefore misses opportunities for short-circuiting.

Ultimately, Recomp-Verify tends to be faster than TLC on benchmarks that have more opportunity for recomposition. We point out that Recomp-Verify^1 is faster than TLC^1 in *every* case where decomposition produced at least four components ($n \geq 4$). The same is true for Recomp-Verify^4 and TLC^4 in all but one case. This observation suggests that the potential benefits of recomposition increase with the number of available components (n).

VIII. Related Work

Compositional verification is a well studied research area. Two widely studied styles of compositional verification are CRA [4], [5], [6], [8], [32], [12], [13], [14], [15], [16] and assume-guarantee reasoning [2], [3], [7], [17], [9], [33], [18], [19], [10], [11], [34], although other styles exist as well [35],

	Recomp-Verify[1]					TLC[1]		Recomp-Verify[4]			TLC[4]	
Name	n	m	k	States	Time	States	Time	States	Time	Strat.	States	Time
tla-consensus-3	1	0	0	4	1s	4	1s	4	1s	S4	4	1s
tla-tcommit-3	1	0	0	34	1s	34	1s	34	1s	S4	34	1s
i4-lock-server-2-2	1	0	0	9	1s	9	1s	9	1s	S4	9	1s
ex-quorum-leader-election-6	2	1	1	117,671	39s	121,111	**4s**	121,111	5s	S4	121,111	**2s**
pyv-toy-consensus-forall-6-6	3	1	1	117,671	33s	121,111	**4s**	121,111	5s	S4	121,111	**2s**
tla-simple-5	1	0	0	723	1s	723	1s	723	1s	S4	723	1s
ex-lockserv-automaton-20	5	1	0	61	**2s**	-	TO	61	**3s**	S3	-	TO
tla-simpleregular-5	1	0	0	2,524	2s	2,524	**1s**	2,524	1s	S4	2,524	1s
pyv-sharded-kv-3-3-3	3	0	0	10,648	5s	10,648	**2s**	10,648	2s	S4	10,648	**1s**
pyv-lockserv-20	5	1	0	61	**1s**	-	TO	61	**2s**	S3	-	TO
tla-twophase-9	4	2	2	145,176	**19s**	10,340,352	9m41s	145,691	**31s**	S1	10,340,352	2m36s
tla-twophase-10	4	2	2	481,550	**1m8s**	-	TO	482,577	**1m36s**	S1	-	TO
tla-twophase-counter-9	5	2	2	145,176	**19s**	-	TO	145,691	**31s**	S1	-	TO
i4-learning-switch-4-3	1	0	-	-	TO	1,344,192	**5m55s**	1,344,192	5m55s	S4	1,344,192	**1m37s**
ex-simple-decentralized-lock-4	2	0	0	20	2s	20	**1s**	20	1s	S4	20	1s
i4-two-phase-commit-7	4	2	2	151,348	**26s**	10,016,384	3m38s	184,112	**27s**	S3	10,016,384	53s
pyv-consensus-wo-decide-4	5	2	1	32,816	**9s**	-	TO	32,953	**10s**	S3	-	TO
pyv-consensus-forall-4-4	6	1	0	33,545	**8s**	-	TO	33,545	**9s**	S3	-	TO
pyv-learning-switch-trans-3	2	1	0	729	**5s**	-	TO	729	**6s**	S1	-	TO
pyv-learning-switch-sym-2	2	1	0	4	2s	1,344	**1s**	1,344	1s	S4	1,344	1s
pyv-sharded-kv-no-lost-keys-3-3-3	2	0	0	9,261	4s	27	**1s**	9,261	2s	S4	9,261	**1s**
ex-naive-consensus-4-4	3	1	1	824	2s	1,001	**1s**	1,001	2s	S4	1,001	**1s**
pyv-client-server-ae-4-2-2	2	1	1	352,145	**42s**	2,039,392	1m36s	352,145	49s	S1	2,039,392	**28s**
pyv-client-server-ae-2-4-2	2	1	1	894,437	2m18s	2,387,032	**1m16s**	2,387,032	1m26s	S4	2,387,032	**22s**
ex-simple-election-6-7	3	1	0	267,590	**1m20s**	2,900,256	3m7s	267,590	1m22s	S3	2,900,256	**54s**
pyv-toy-consensus-epr-8-3	3	1	1	65,543	1m1s	70,903	**6s**	70,903	7s	S4	70,903	**2s**
ex-toy-consensus-8-3	2	1	1	65,543	57s	70,903	**5s**	70,903	6s	S4	70,903	**2s**
pyv-client-server-db-ae-2-3-2	5	4	4	188,158	**12s**	1,394,368	1m1s	188,799	**15s**	S1	1,394,368	18s
pyv-client-server-db-ae-4-2-2	5	1	1	356,706	**1m23s**	3,624,960	2m48s	356,706	1m40s	S1	3,624,960	**44s**
pyv-firewall-5	2	0	0	56,072	9s	56,072	**2s**	56,072	3s	S4	56,072	**1s**
ex-majorityset-leader-election-5	3	1	-	-	TO	166,306	**15s**	166,306	17s	S4	166,306	**5s**
pyv-consensus-epr-4-4	6	2	1	7,018	**3s**	-	TO	7,221	**5s**	S3	-	TO
mldr-2	1	0	-	-	TO	-	TO	-	TO	-	-	TO

Fig. 6: Run time comparison between Recomp-Verify and TLC. The superscripts for each tool indicates how many threads are allocated to a trial. The fastest times for each experiment are bolded. The "Strat." column denotes the fastest strategy.

[36]. Of these works, the ones most closely related to this paper automate decomposition for verification. Metzler et a. [18] and Cobleigh et al. [17] decompose systems into two components, after which they apply L* style learning [37] to infer assumptions for assume-guarantee style compositional verification. Nam et al. [19] use a similar strategy, but consider multi-way decomposition and verification. While these works report limited success, we are able to find efficient verification problems via recomposition.

Our work also relates to program slicing [20], [21] and cone of influence reduction [38], both of which are techniques for static specification reduction. These two techniques soundly reduce the state variables needed for model checking by analyzing a variable dependency graph. Our work includes static specification reduction by allowing partial recomposition maps, as described in Sec. V.

In the TLA$^+$ ecosystem, TLC [23] is the most well-known model checker. Apalache [39], [40] is an alternate model checker that internally relies on SMT solvers. Apalache supports bounded model checking and verification with inductive invariants–two techniques that are outside the scope of comparison for our tool. The TLA$^+$ Proof System (TLAPS) [41] provides an alternative to model checking TLA$^+$. TLAPS proofs are manually constructed, but automatically verified by dispatching proof obligations to SMT solvers. Endive [31] is a tool that automatically infers inductive invariants for TLA$^+$ specifications, which then may be checked using a TLAPS proof.

IX. Limitations and Future Work

In Sec. VII-C, we show that the effectiveness of our approach is tied to the number of C_i components. In future work, we plan to investigate methods to make decomposition more granular, as well as decomposing *properties*. As the number of components increase, we also plan to improve our methods for finding efficient recomposition maps. For example, we plan to improve our parallel technique so that different threads can share intermediate work to save time and memory.

In this paper, we focus on using recomposition for explicit-state model checking. However, recomposition may also be effective for other compositional verification tasks. For example, we plan to investigate whether recomposition can be used in combination with non-explicit verification techniques, e.g. using SMT solvers. We also plan to investigate whether recomposition can be used for efficient counter-example detection.

Acknowledgements

The authors are most grateful to Changjian Zhang, Andy Hammer, and the anonymous reviewers for their helpful comments on earlier versions of this paper. This project was supported by the U.S. NSF Award #2144860.

References

[1] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu, *Compositional Reasoning*. Cham: Springer International Publishing, 2018, pp. 345–383. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_12

[2] R. Alur, P. Madhusudan, and W. Nam, "Symbolic compositional verification by learning assumptions," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 548–562.

[3] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, "Learning minimal separating dfa's for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 31–45.

[4] S. C. Cheung and J. Kramer, "Compositional reachability analysis of finite-state distributed systems with user-specified constraints," in *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 140–150. [Online]. Available: https://doi.org/10.1145/222124.222149

[5] ——, "Context constraints for compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, p. 334–377, oct 1996. [Online]. Available: https://doi.org/10.1145/235321.235323

[6] ——, "Checking safety properties using compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 1, p. 49–78, jan 1999. [Online]. Available: https://doi.org/10.1145/295558.295570

[7] J. M. Cobleigh, D. Giannakopoulou, and C. S. PĂsĂreanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 331–346.

[8] S. Graf and B. Steffen, "Compositional minimization of finite state systems," in *Proceedings of the 2nd International Workshop on Computer Aided Verification*, ser. CAV '90. Berlin, Heidelberg: Springer-Verlag, 1990, p. 186–196.

[9] A. Gupta, K. L. McMillan, and Z. Fu, "Automated assumption generation for compositional verification," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 420–432.

[10] C. Păsăreanu, D. Giannakopoulou, M. Bobaru, J. Cobleigh, and H. Barringer, "Learning to divide and conquer: Applying the l*algorithm to automate assume-guarantee reasoning," *Formal Methods in System Design*, vol. 32, pp. 175–205, 06 2008.

[11] C. S. Păsăreanu and D. Giannakopoulou, "Towards a compositional spin," in *Model Checking Software*, A. Valmari, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 234–251.

[12] K. Sabnani, A. Lapone, and M. Uyar, "An algorithmic procedure for checking safety properties of protocols," *IEEE Transactions on Communications*, vol. 37, no. 9, pp. 940–948, 1989.

[13] K.-C. Tai and P. Koppol, "Hierarchy-based incremental analysis of communication protocols," in *1993 International Conference on Network Protocols*, 1993, pp. 318–325.

[14] H. Zheng, "Compositional reachability analysis for efficient modular verification of asynchronous designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 329–340, 2010.

[15] ——, "Local state space construction for compositional verification of concurrent systems," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, ser. SPIN 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 11–19. [Online]. Available: https://doi.org/10.1145/2632362.2632366

[16] H. Zheng, Z. Zhang, C. J. Myers, E. Rodriguez, and Y. Zhang, "Compositional model checking of concurrent systems," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1607–1621, 2015.

[17] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 97–108. [Online]. Available: https://doi.org/10.1145/1146238.1146250

[18] B. Metzler, H. Wehrheim, and D. Wonisch, "Decomposition for compositional verification," in *Formal Methods and Software Engineering*, S. Liu, T. Maibaum, and K. Araki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 105–125.

[19] W. Nam, P. Madhusudan, and R. Alur, "Automatic symbolic compositional verification by learning assumptions," *Form. Methods Syst. Des.*, vol. 32, no. 3, p. 207–234, jun 2008. [Online]. Available: https://doi.org/10.1007/s10703-008-0055-8

[20] I. Brückner and H. Wehrheim, "Slicing an integrated formal method for verification," in *Formal Methods and Software Engineering*, K.-K. Lau and R. Banach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 360–374.

[21] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, p. 446–452, jul 1982. [Online]. Available: https://doi.org/10.1145/358557.358577

[22] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002. [Online]. Available: https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/

[23] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications," in *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66.

[24] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, p. 133–160, mar 2006. [Online]. Available: https://doi.org/10.1145/1132863.1132867

[25] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[26] D. Giannakopoulou, C. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, 2002, pp. 3–12.

[27] W. J. Yeh and M. Young, "Compositional reachability analysis using process algebra," in *Proceedings of the Symposium on Testing, Analysis, and Verification*, ser. TAV4. New York, NY, USA: Association for Computing Machinery, 1991, p. 49–59. [Online]. Available: https://doi.org/10.1145/120807.120812

[28] R. Milner, *Communication and Concurrency*, ser. Ph/AMA Series in Marketing. Prentice Hall, 1989. [Online]. Available: https://books.google.com/books?id=S5UZAQAAIAAJ

[29] I. Dardik, A. Porter, and E. Kang, "Recomposition: A new technique for efficient compositional verification," 2024. [Online]. Available: https://arxiv.org/abs/2408.03488

[30] "Recomp-verify research prototype model checker for tla+," https://github.com/cmu-soda/recomp-verify/tree/FMCAD24, accessed: 2024-08-16.

[31] W. Schultz, I. Dardik, and S. Tripakis, "Plain and simple inductive invariant inference for distributed protocols in tla+," in *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2022, pp. 273–283.

[32] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro, "Winston: A tool for hierarchical design and simulation of concurrent systems," in *Specification and Verification of Concurrent Systems*, C. Rattray, Ed. London: Springer London, 1990, pp. 140–152.

[33] C. Jones, "Specification and design of (parallel) programs." vol. 83, 01 1983, pp. 321–332.

[34] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 123–144.

[35] H. Barringer, D. Giannakopoulou, and C. Pasareanu, "Proof rules for automated compositional verification through learning," 02 2003.

[36] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *Automated Technology for Verification and Analysis*, S. S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 64–79.

[37] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, p. 87–106, nov 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[38] E. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking, second edition*, ser. Cyber Physical Systems Series. MIT Press, 2018. [Online]. Available: https://books.google.com/books?id=qJl8DwAAQBAJ

[39] I. Konnov, J. Kukovec, and T.-H. Tran, "Tla+ model checking made symbolic," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360549

[40] R. Otoni, I. Konnov, J. Kukovec, P. Eugster, and N. Sharygina, "Symbolic model checking for tla+ made faster," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds. Cham: Springer Nature Switzerland, 2023, pp. 126–144.

[41] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "Tla+ proofs," *Proceedings of the 18th International Symposium on Formal Methods (FM 2012), Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science*, vol. 7436, pp. 147–154, January 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/tla-proofs/

Formal Methods in Computer-Aided Design 2024

Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages

Shuvendu K. Lahiri
Microsoft Research, Redmond, USA
shuvendu@microsoft.com

***Abstract*—Verification-aware programming languages such as Dafny and F* provide means to formally specify and prove properties of a program. Although the problem of checking an implementation against a specification can be defined mechanically, there is no algorithmic way of ensuring the correctness of the *user-intent formalization for programs* — that a specification adheres to the user's intent behind the program. This is because intent or requirement is expressed *informally* in natural language and the specification is a formal artefact. However, the advent of large language models (LLMs) has made tremendous strides bridging the gap between informal intent and formal program implementations in the last couple of years, driven in large parts due to benchmarks and automated metrics to evaluate different techniques.**

Recent work has developed a framework for evaluating and benchmarking the *user-intent formalization* problem for mainstream programming languages [12]. However, as we argue in this paper, such an approach does not readily extend to verification-aware languages that support rich specifications (using quantifiers and ghost variables) that cannot be evaluated through dynamic execution. Previous work also required generating program mutants using LLMs to create the benchmark. We advocate an alternate, perhaps simpler approach of *symbolically testing specifications* to provide an intuitive metric for evaluating the quality of specifications that can be easily instantiated with most verification-aware languages. We demonstrate that our automated metric agrees closely on a human-labeled dataset of Dafny specifications for the popular MBPP code-generation benchmark, yet demonstrates cases where the human labeling is not perfect. We also outline formal verification challenges that need to be addressed to apply the technique more widely. We believe our work provides a stepping stone to enable the establishment of a benchmark and research agenda for the problem of user-intent formalization for programs.

***Index Terms*—formal verification, specifications, large language models**

I. Introduction

Formal verification is only as good as the specification it verifies. A formal specification unambiguously defines the (possibly partial) intent behind a program, often in an declarative manner. Although there has been decades of research in advancing the state-of-the-art in automating the problem of verifying an implementation against a specification, relatively less attention has been spent on how to aid the generation and evaluation of specifications or formal requirements. It is well-known that the lack of formal specifications is a significant impediment to deployment of formal verification in production code [10]. On the other hand, although a software is often accompanied by informal intent expressed in natural language comments and API documentation, these intents are seldom formally enforced on the underlying implementation.

Large language models (LLMs) have recently demonstrated potential to bridge the gap between informal intent and formal artefact such as code [9], [5], by performing code generation from natural language. Such progress has been spurred in large parts through the creation of crowd-sourced benchmarks such as HumanEval [9] and Mostly Basic Python Programs (MBPP) [5] with automated program-semantics based metrics (such as tests), unlike NLP metrics such as BLEU scores [25]. For code generation, the quality of an implementation generated from informal intent is measured through the set of *hidden* validation tests. We term the tests as ”hidden” since they are not available to the model at the time of code generation. The benchmarks also provide a reference code in Python for each problem, which are also hidden from the code generation model. These approaches rely heavily on the presence of a high-quality set of validation tests to exercise most corner cases. These benchmarks allow a community to evaluate their models and techniques on a common set of benchmarks automatically and measure progress. One can hope that the establishment of similar benchmarks and metrics for specification generation (decoupled from code generation) can enable synthesizing useful specifications from informal intent in practice.

Motivated by such a need, Endres *et al.* [12] describe the problem of generating formal declarative specifications (namely, method postconditions) from informal intent using LLMs and automated metrics for evaluating them — we refer to this as *user-intent formalization* problem in this paper. For user-intent formalization, Endres *et al.* re-purpose code-generation benchmarks (such as HumanEval) and introduce two metrics (a) *correctness* and (b) *completeness*, with respect to the set of (hidden) validation tests and the (hidden) reference code. Correctness captures that the specification satisfies the reference code for all the validation tests; completeness captures the strength of the specification to discriminate against buggy mutations of the reference code (under the set of tests). Further, these code mutants are generated through LLMs and are grouped by the subset of tests that fail them. They demonstrate that these automatic metrics closely resemble the quality of specifications as determined through manual analysis. As with code generation, the approach relies on the

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_19

presence of a high-quality test-suite.

A. Summary

In this paper, we investigate creating benchmarks (dataset and associated automatic metrics) for user-intent formalization for *verification-aware languages* such as F* [27], Dafny [21] and Verus [20]. A verification-aware language supports a rich program logic for expressing specifications and offers the ability to verify them statically using automated theorem provers. However, for most non-trivial programs, the verification requires manually decomposing the problem through the use of ghost variables, intermediate lemmas, invariants and assertions.

We focus on the problem of evaluating specifications automatically. We discuss why prior approaches do not readily apply to evaluating specifications. Then we propose the use of automatic program verification to symbolically ”test” the specifications to determine their quality. We have developed a prototype and applied it to a dataset of Dafny specifications from prior work [23]. We demonstrate that the quality of the specifications obtained through the automated means aligns well with the human-labeling of the specifications in most cases. Finally, we describe the unique challenges (such as quantifier instantiation) that need to be addressed to apply the technique to a larger class of problems.

B. Running example

Consider a snippet of the JSON specification of an example from the MBPP dataset for code generation from natural language in Python [9].

```
"prompt": "Write a function to find the shared elements
    from the given two lists.",
"code": "def similar_elements(test_tup1, test_tup2):\n
    ....",
"test_list": [
   "assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10)
       )) == set((4, 5))",
   ...
]
```

The dataset comes with a set of problems, each containing a natural language prompt prompt, the reference code code as well a set of 3 tests in test_list . Misu *et al.* [23] port these requirements to Dafny (call it MBPP-DFY), including slight change to the prompt, explicitly representing the Dafny method signature (with a slightly changed name), as well as the test cases [1]. [1]

```
"task_description": "Write a method in Dafny to find the
    shared elements from the given two array.",
"method_signature": "method similarElements (arr1:array<int
    >, arr2:array<int>) returns (res: array<int>)",
"test_cases": {
    "test_1": "var a1:= new int[] [3, 4, 5, 6];\nvar a2:=
        new int[] [5, 7, 4, 10];\nvar e1:= new int[] [4,
        5];\nvar res1:=similarElements(a1,a2);\nassert
        arrayEquals(res1,e1);",
      ...
}
```

[1]We take the snapshot of the repository at COMMIT a57ce24.

Misu *et al.* use GPT-4 [4] and other language models (with sophisticated prompting) to generate a pair of Dafny specification and implementation, and retain generations where the generated specification is provable for the generated implementation. Below we show the specification component of the GPT-4 generated Dafny artefact for this example (with slightly altered method name and signature over JSON) [3].The two postconditions (marked with ensures) define the specification and use an auxiliary predicate InArray.

```
predicate InArray(a: array<int>, x: int)
reads a
{ exists i :: 0 <= i < a.Length && a[i] == x }

method SharedElements(a: array<int>, b: array<int>)
              returns (result: seq<int>)
  ensures forall x :: x in result ==>
                     (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==>
                     result[i] != result[j]
{
   ....Dafny implementation...
}
```

Although the implementation satisfies the specification, there is no evidence that the specification indeed captures what the user intended (and specified implicitly in the hidden test cases). The authors perform *manual* review of the specifications and mark them as either {WRONG_SPEC, WEAK_SPEC, STRONG_SPEC} to denote if the specification is respectively, inconsistent, weakly or strongly consistent with the intent expressed in the natural language and tests [2]. Therefore, although this work addresses generating verified Dafny programs from informal intent, it relies completely on a human to ensure that the specification matches the intent. In particular, the framework allows for the Dafny code to only satisfy a vacuous specification (such as ensures `true`) or worse, an incorrect one. This aspect renders this dataset unsuitable as an automated benchmark for specification generation in verification-aware languages. In other words, if a language model generates a specification that is non-equivalent to the ground-truth specification labeled by the user, we cannot determine if it is incorrect without requiring a user. This also makes the evaluation subjective. Since verification of (generated) Dafny code is only as good as the specification, this makes the dataset unsuitable for an automated benchmark for verified code generation as well.

C. Proposal: Symbolically testing specifications

Our objective in this work is to define metrics that can be automatically evaluated to determine the quality of a specification. At the same time, we would also like to establish that these metrics correspond well to what a manual labeling would establish. Further, we decouple the problem of specification generation from code generation, to be able to (a) use the specifications to find bugs in underlying or generated code [12], or (b) use the validated specifications to refine the prompts for code generation.

We first argue that the none of the prior approaches are readily applicable to yield an automatic metric: (i) First, running the tests on the implementation without specifications will not rule out vacuous specifications. (ii) Second, one cannot apply the approach of Endres *et al.* for rich specifications that contain ghost state or constructs that cannot be evaluated at runtime (such as a universal/existential quantifier). (iii) One can ignore tests and attempt to statically verify the synthesized specification directly against the hidden reference code. This can still allow vacuous specifications such as `true`. Further, it is unlikely that such verification would be automatic for non-trivial specifications given the need to infer intermediate lemmas and invariants. (iv) One can instead provide a hidden reference specification and check the candidate specification for semantic equivalence. However, such a metric would be too strict as it would not be able to distinguish weak and vacuous specifications (such as `true`) from strong (yet incomplete) specifications.

Next, we propose a method for evaluating user-intent formalization for verification-aware languages that leverages the validation tests and symbolic verification capabilities of these languages. Given a set of tests consisting of a set of *consistent* input-output pairs, we define the correctness and completeness metrics purely over the tests, without the need for the reference code. Of course, just as in the case of code generation, the technique for determining specification quality assumes a high-quality set of validation tests.

Consider a method with signature m(x):y denoting the name m, input parameters x and output parameters y, a candidate postcondition/summary specification ϕ(x, y) and a set of input-output tests T.

1) Correctness: A postcondition ϕ(x, y) is correct (or sound) with respect to T if it is consistent with all the input-output pairs in T. In other words, for each $(i, o) \in T$ the following Hoare-triple [15] holds.

$$\models \{\texttt{true}\}\ \mathrm{x} := i;\ \mathrm{y} := o;\ \{\phi(\mathrm{x}, \mathrm{y})\}$$

2) Completeness: The completeness measure for a specification ϕ given T is the fraction of output mutations of the tests in T that ϕ is inconsistent with. Let $T_1 \doteq \bigcup_{(i,o)\in T} \bigcup_{o' \neq o} \{(i, o')\}$ be a finite set of mutations of T that mutate the output values for the given inputs. In this work, we restrict the set of mutants per input i to a fixed number (5 for this paper). Let $T_2 \subseteq T_1$ be the largest subset such that for each $(i, o') \in T_2$, the following Hoare-triple *does not* hold:

$$\not\models \{\texttt{true}\}\ \mathrm{x} := i;\ \mathrm{y} := o';\ \{\phi(\mathrm{x}, \mathrm{y})\}$$

Then the completeness measure of ϕ with respect to T is $|T_2|/|T_1|$ (this is inspired by kill-set in *mutation-testing* literature [16]). For interested readers, we also contrast these with a plausible proposal in Appendix B.

Let us demonstrate an implementation of these Hoare triples as Dafny programs for our running example. Consider the first test (see JSON input in Section I-B) that asserts that SharedElements should return the set $\{4, 5\}$ for input arrays $[3, 4, 5, 6]$ and $[5, 7, 4, 10]$. For correctness against this test, we create the following Dafny program by providing a definition of SharedElements that performs the soundness check:

```
predicate InArray(a: array<int>, x: int)
reads a
{  exists i :: 0 <= i < a.Length && a[i] == x }

method SharedElements(a: array<int>, b: array<int>)
              returns (result: seq<int>)
  ensures forall x :: x in result ==>
                        (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==>
                        result[i] != result[j]
{
  var a1 := new int[] [3, 4, 5, 6];
  var a2 := new int[] [5, 7, 4, 10];
  assume {:axiom} a[..a.Length] == a1[..a1.Length];
  assert a[0] == a2[0] && .... && a[3] == a2[3];
  assume {:axiom} b[.. b.Length] == a2[..a2.Length];
  assert b[0] == a2[0] && .... && b[3] == a2[3];
  result := [4, 5];
}
```

The Dafny program above is identical to the program in Section I-B, except for the body of SharedElements specified within the curly braces $\{\ldots\}$. Instead of the original implementation of the method, we model the Hoare-triple for correctness described above. The precondition of the Hoare-triple (`true`) translates to assume true which is dropped. The input assignment $\mathrm{x} := i;$ is modeled as assignments of the input i to temporary variables a1, a2, followed by constraining the actual parameters a, b respectively. For Dafny, this amounts to saying that two arrays a (respectively, b) and a1 (respectively, a2) are equal on all elements up to their lengths (which implies that the lengths are identical as well). The redundant asserts are needed for the verifier to trigger the quantifiers used in InArray to enable the proof. Finally, the output parameter result is assigned one of the expected values ([5,4] is also an acceptable value). The Dafny program symbolically checks (using Satisfiability Modulo Theories solvers [6]) that the specification (provided by the ensures statements) holds for the specific input and output.

However, the above verification only proves that the specification is *correct* for the test; a vacuous specification ensures true would also be verified. For completeness, we mutate the output value in result in several ways and check that the verification fails. In this case, mutating the value in result to [6] would fail the first postcondition, since 6 is not present in both the input arrays.

For this program, the above specification is marked as STRONG_SPEC by the authors of MBPP-DFY. However, our automated test harness coupled with mutation discovers that the specification is *incomplete* (score 0.6). When we check the above program with either of the two mutations result := [4] or result := [5], the specification still verifies. This is because the implication ==> in the first postcondition only checks that values in result is present in both the arrays; it does not check that all such common values are present in result! Thus our automated metric is able to assign the above specification a lower score than the full functional specification for this example where the ==> is replaced by <==>.

Note: One may caution against using a verification failure (as used for the completeness checks) as a means to show that the corresponding Hoare-triple does not hold. In other words, given that deductive verifiers are sound, but not necessarily complete, a verification failure is typically interpreted as an unknown outcome. However, note that we check for completeness for a specification ϕ against an input-output example (i, o') only if ϕ is *proved correct* for all the tests in *T* (including the original test (i, o)). Since the two verification conditions differ only in a concrete value (between o and o'), we conjecture that the underlying reasoning (including quantifiers instantiated) would be quite identical in most cases, and the verification failure strongly indicates that the Hoare-triple does not hold.

II. Implementation and evaluation

We report ongoing work in implementing the two metrics for the MBPP-DFY dataset mentioned earlier in Section I-B. Our tool (a 400 line Python script) consumes the method signature, test cases and the candidate specification from the JSON and Dafny files, and creates Dafny programs for verifying the correctness and completeness for each test. It then invokes the Dafny verifier, and reports the aggregate correctness as well completeness score for each specification, averaged over the different test cases; it finally compares the metrics against the labels provided by authors. Of the 153 problems with specifications (a subset written by humans), we have managed to apply our tool to check 64 of the specifications (at the time of writing). In other words, for these 64 examples, our tool is able to verify the soundness of the specifications over the set of validation tests. We have released the scripts, dataset and outputs at the website https://github.com/microsoft/nl-2-postcond.

A. Mutating values

Each of these examples contain 3 test cases in Dafny format, and we consider up to 5 distinct mutants of the output values for each test. We currently have a simple mutation scheme for the output values. For Booleans, we flip the value between `true` and `false`. For integers, we choose a random value between 1 and 10, and randomly add or subtract it. For strings, we choose a random character and either replace one of the characters or append it to the string. For arrays (we only restrict to integer arrays and sequences), we choose between dropping an element or inserting a random value at a randomly chosen index. For our running example of SharedElements, this allows us to create a mutant test case with the array value $[4]$ that demonstrates that the GPT-4 generated specification (marked as STRONG_SPEC) is not the most precise specification for this problem.

B. Results

We briefly report some details of the evaluation, and outline the challenges to handle the remaining examples.

We find that for large majority of examples, the manual labels align with the metrics computed by our tool. That is, for the incorrect specifications, our tool reports a verification failure for correctness check, and report a high completeness score (usually > 0.66) for precise specifications. In other words, barring a few exceptions below, all the specifications that are marked STRONG_SPEC have a completeness score above 0.66, and all the WEAK_SPEC have a score below 0.66.

This provides evidence that the automated metrics serve as a good proxy for the user label. One such incorrect specification generated by GPT-4 (and correctly labeled) is the problem of RemoveDuplicates ("task_id" 572) (also the motivating example by Endres *et al.* [12] demonstrating ambiguity of natural language) where the task is to remove all elements with duplicates, but GPT-4 interprets it as the problem of retaining a single copy of each value. Similarly, for countSubstrings ("task_id" 61), that counts the number of substrings whose sum equals their length, our completeness check gives a low score to the specification (correctly labeled WEAK_SPEC) that only ensures count is a non-negative number.

In addition to the SharedElements ("task_id" 2), we found at least 2 more cases where the specification (labeled STRONG_SPEC) is weaker than the most precise one. These include the cases for maxAbsDiff ("task_id" 145), and removeElements ("task_id" 161). maxAbsDiff is expected to compute the maximum difference between any two elements in the list; the provided specification only ensures that the result upper bounds the difference between any two elements, but is not the precise bound. removeElements is expected to remove the elements in one array from another; the specification only ensures that any element in the result array is present in the first array and not the second but fails to ensure that the result array is the precise difference (can be ensured by changing `==>` by `<==>`).

Interestingly, we also discovered a few cases where the correctness check failed for specifications labeled STRONG_SPEC. These include examples with "task_id"s 234, 240, and 445. For these examples, the authors had accidentally introduced bugs while copying from the test cases in Python. For instance, for "task_id" 234 that computes the cube of a number, the authors mistakenly copied the value 25 instead of 125 for the cube of 5. We noticed that a couple of these errors have been fixed in the latest commit at the time of writing (fb4f53e), but a few still remain (e.g., 445 where the value 32 is replaced by 31).

Our preliminary results demonstrates that the automatic metrics not only helps to add objectivity to the manual labeling, argues for having a quantitative metric for completeness (instead of a Boolean WEAK_SPEC vs STRONG_SPEC), but also clearly demonstrates the potential to serve as a metric for evaluating specifications for a benchmark.

C. Limitations

Our implementation currently has a few limitations that precludes analyzing all the 153 specifications correctly. A large class of them stem from inaccurate parsing through a simple regex based parsing of the Dafny files. We expect these (e.g., handling of 2D arrays) to be addressed as we improve the parsing through Dafny AST-based analysis. However, a few fundamental challenges (related to failure to perform proofs of correct specifications) remain that requires further investigation. We note a couple of them here:

a) Recursive predicates.: Specifications for tasks such as 105 (counting number of true Booleans in an array) require the use of recursive functions countTo that expresses the count of an array in terms of countTo of the tail of the array. Dafny uses a notion of "fuel" to control the level of unfolding of such predicates, which suffices for inductive proofs. However, for concrete arrays in test cases, these recursive functions need to be unrolled proportional to the length of the array.

b) Quantifier instantiation.: Specifications for tasks such such as 3 (that classifies a number as a non-prime) uses existential quantifiers. For example, checking a number n as non-prime requires checking for the existence of a smaller number k that divides n. Unlike their usage in inductive proofs where a loop establishes the inductive hypothesis to only instantiate the quantifier on a few expressions, testing a symbolic specification against a concrete value is sometimes non-trivial. For example, to show that the value 97 is a prime, the quantifier needs to be instantiated on all values up to 96.

We have already automated a few failed proofs due to quantifier instantiation by adding intermediate redundant assertions equating two arrays on every index (see assert statements in the body of SharedElements in Sec I-B). In future work, we plan to automate the inference of fuel amount and quantifier instantiation witnesses when ranging over the indices of an array.

III. Related work

Although the literature on specification mining is fairly rich, it has historically focused on techniques that infer specifications of an implementation through dynamic [13] and recently neural techniques [26]. The objective of these techniques is to learn invariants from a few runs that generalize to unseen executions. Static [14], [19], [24] and more recently neural approaches [28], [17] have also been applied to the problem (inductive) invariant generation to aid program proofs. Finally, recent works have explored the generation of programs, specifications and proofs from natural languages [23], [29], [22]. On the other hand, work on user-intent formalization (i.e., translating natural language comments to specifications) [7], [12] synthesize the intended specification, even in the absence of an implementation. The problem of generating *test oracle* assertions have also been investigated [11], but they only apply to single input or test prefix. TiCoder [18] introduced the term "user-intent formalization" over test cases for interactive code generation, and alluded to metrics for measuring quality of weak specifications (namely, tests) in terms of correctness (user acceptance) and completeness (prunes buggy codes). The work on *autoformalization* [30] for translating natural language comments to mathematical theorems is closest to our work. However, these techniques do not use semantic checks (such as tests and symbolic verification) to ensure the quality of the generated formal mathematical statements; instead they leverage textual similarity metrics such as BLEU.

IV. Conclusion

In this paper, we motivate the problem of evaluating user-intent-formalization for verification-aware languages. We demonstrate that the idea of symbolically testing specifications against validation tests can provide an automated metric for a benchmark. We plan to curate a benchmark with a large fraction of examples from MBPP-DFY dataset using the above metric. We hope such benchmarks will accelerate the research on specification generation from informal intent in verification-aware languages. This when coupled with work on program and proof synthesis (given a specification) can greatly lower the cost required to create formally verified modules in the future [8].

Appendix A
Extended version of the JSON examples

```
"prompt": "Write a function to find the shared elements
     from the given two lists.",
"code": "def similar_elements(test_tup1, test_tup2):\n
     ....",
"test_list": [
    "assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10)
        )) == set((4, 5))",
    "assert set(similar_elements((1, 2, 3, 4),(5, 4, 3, 7))
        ) == set((3, 4))",
    "assert set(similar_elements((11, 12, 14, 13),(17, 15,
        14, 13))) == set((13, 14))"
]
```

```
"task_description": "Write a method in Dafny to find the
     shared elements from the given two array.",
"method_signature": "method similarElements (arr1:array<int
     >, arr2:array<int>) returns (res: array<int>)",
   "test_cases": {
      "test_1": "var a1:= new int[] [3, 4, 5, 6];\nvar a2:=
           new int[] [5, 7, 4, 10];\nvar e1:= new int[] [4,
           5];\nvar res1:=similarElements(a1,a2);\nassert
          arrayEquals(res1,e1);",
      "test_2": "var a3:= new int[] [1, 2, 3, 4];\nvar a4:=
           new int[] [5, 4, 3, 7];\nvar e2:= new int[] [3,
          4];\nvar res2:=similarElements(a3,a4);\nassert
          arrayEquals(res2,e2);",
      "test_3": "var a5:= new int[] [11, 12, 14, 13];\nvar
          a6:= new int[] [17, 15, 14, 13];\nvar e3:= new
          int[] [13, 14];\nvar res3:=similarElements(a5,a6)
          ;\nassert arrayEquals(res3,e3);"
    }
```

Appendix B
Alternate proposal

A reader may wonder if the following check achieves a similar objective for checking completeness:

$$\models \{\mathrm{x} == i \wedge \phi(\mathrm{x}, \mathrm{y})\} \text{ skip } \{\mathrm{y} == o\}$$

There are two issues with this formulation: (a) this only provides a Boolean metric that only rewards a precise specification that constrains y to a unique value o. For example, it would award the strong (yet imprecise) specifications of SharedElements to 0. Secondly, (b) this formulation is also too strong when the precise functional specification allows for non-determinism in the output. Consider the running example of SharedElements, where the specification allows for the output to be one of either $\{[4, 5], [5, 4]\}$ for the given inputs.

References

[1] mbpp-san-dfy-228-all-task-test.json. https://github.com/Mondego/dafny-synthesis/blob/a57ce24/MBPP-san-DFY-228/mbpp-san-dfy-228-all-task-test.json.

[2] rq3-dynamic-few-shot-prompting-gpt-4-temp_0.5-verified-unverified-tagged.json. https://github.com/Mondego/dafny-synthesis/blob/a57ce24/RQs/RQ3-%5BDynamic-Few-Shot-Prompting%5D/rq3-dynamic-few-shot-prompting-GPT-4-temp_0.5-verified-unverified-tagged.json.

[3] task_id_2.dfy. https://github.com/Mondego/dafny-synthesis/blob/a57ce24/MBPP-DFY-153/test/task_id_2.dfy.

[4] Gpt-4 technical report, 2024.

[5] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.

[6] C. W. Barrett, L. M. de Moura, S. Ranise, A. Stump, and C. Tinelli. The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In S. Barner, I. G. Harris, D. Kroening, and O. Raz, editors, *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010*, volume 6504 of *Lecture Notes in Computer Science*, page 3. Springer, 2010.

[7] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 242–253, 2018.

[8] S. Chakraborty, G. Ebner, S. Bhat, S. Fakhoury, S. Fatima, S. Lahiri, and N. Swamy. Towards neural synthesis for smt-assisted proof-oriented programming, 2024.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

[11] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2130–2141, 2022.

[12] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri. Can large language models transform natural language intent into formal method postconditions? In *Proceedings of the Foundations of Software Engineering 2024 (FSE'24)*, 2024.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.

[14] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*. Springer Berlin Heidelberg, 2001.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[16] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[17] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma. Finding inductive loop invariants using large language models, 2023.

[18] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*, 2022.

[19] S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. Intra-module inference. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, pages 493–508, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[20] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023.

[21] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[22] C. Loughridge, Q. Sun, S. Ahrenbach, F. Cassano, C. Sun, Y. Sheng, A. Mudide, M. R. H. Misu, N. Amin, and M. Tegmark. Dafnybench: A benchmark for formal software verification, 2024.

[23] I. M. Md Rakib Hossain Misu, Cristina V. Lopes and J. Noble. Towards ai-assisted synthesis of verified dafny methods, 2024.

[24] O. Padon, J. R. Wilcox, J. R. Koenig, K. L. McMillan, and A. Aiken. Induction duality: primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[26] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin. Can large language models reason about program invariants? 2023.

[27] J. Protzenko, J. K. Zinzindohoue, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in f*. In *22nd International Conference on Functional Programming (ICFP 2017)*. ACM SIGPLAN, May 2017.

[28] X. Si, A. Naik, H. Dai, M. Naik, and L. Song. Code2inv: A deep learning framework for program verification. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020.

[29] C. Sun, Y. Sheng, O. Padon, and C. Barrett. Clover: Closed-loop verifiable code generation, 2024.

[30] Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C. Szegedy. Autoformalization with large language models, 2022.

Towards Verification Modulo Theories of asynchronous systems via abstraction refinement

Gianluca Redondi
Fondazione Bruno Kessler
Trento, Italy
gredondi@fbk.eu

Alessandro Cimatti
Fondazione Bruno Kessler
Trento, Italy
cimatti@fbk.eu

Alberto Griggio
Fondazione Bruno Kessler
Trento, Italy
griggio@fbk.eu

***Abstract*—This paper introduces a new algorithm designed to verify safety properties for asynchronous compositions of symbolic transition systems. The approach combines under-approximation and over-approximation: on one side, it zooms in on a selected set of components, while forcing the remaining ones to stutter; on the other, the selected components are individually abstracted and re-composed. This strategy can be advantageous for scenarios involving large numbers of components, where only a small subset of key components allows to produce the right invariants for the system. We detail the application of our algorithm to a class of parameterized symbolic transition systems, by using a form of slicing as an abstraction. Our experimental results, although preliminary, show the potential of the approach.**

***Index Terms*—Compositional Approach, CEGAR, Parameterized Systems**

I. Introduction

This paper focuses on the problem of asynchronous verification, where multiple systems with shared variables undergo transitions independently. The problem is studied in the literature across various formalisms. In this paper, we adopt the formalism of Verification Modulo Theories [3], using symbolic transition systems defined by formulae in SMT. We describe an algorithm that we aim to use in a verification project related to railway interlocking logic [1], [5], which is characterized by numerous components and a large number of variables. However, the verification of safety properties may necessitate examining only a select few of these components. Additionally, many variables within these components are not integral to managing safety operations. Hence, our algorithm seeks to abstract numerous system variables while potentially concentrating on the pertinent components.

Initially, we outline the algorithm in a generic scenario, not focusing on particular abstraction or class of systems. Then, we narrow our focus to a more concrete use case. We define a class of transition systems capable of modeling parameterized systems and instantiate the aforementioned approach by providing specific procedures for abstraction and refinement. Such a scenario can model, for example, the interlocking logic we are interested in.

The authors acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU, and of the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

We have developed a prototype of the algorithm and tested it on some artificial benchmarks and a simplified case study related to interlocking logic. The outcomes are encouraging, suggesting that this algorithm could also perform effectively on the entire logic system once we acquire the comprehensive system descriptions.

The paper is organized as follows: Section 2 provides the necessary background, and it studies the connections between abstraction and composition. Section 3 details the procedure for the verification of systems defined via asynchronous composition. Section 4 specializes the algorithm in the case of parameterized systems and presents the experimental evaluation. Finally, Section 5 describes our conclusions and outlines avenues for future work.

II. Background

A. Preliminaries

Our models of computation are symbolic transition systems, i.e. triples of the form $(X, I(X), T(X, X'))$, where X is a set of variables, called the *state variables* of the system, and $I(X)$, $T(X, X')$ are formulae over some theory $\mathcal{T}$. Given a model $\mathcal{M}$ for $\mathcal{T}$, a state s is a valuation of the state variables X in the universe of $\mathcal{M}$. A state is initial iff it is a model of $I(X)$, i.e., $s \models I(X)$. A pair of states s, s' denotes a transition iff $s, s' \models T(X, X')$. A state s is reachable iff there exists a path π such that $\pi[i] = s$ for some i.

A formula $\phi(X)$ is an invariant of the transition system $C = (X, I(X), T(X, X'))$ iff it holds in all the reachable states. Following the standard model checking notation, we denote this as $C \models \phi(X)$. We say that ϕ is inductive for C if $I(X) \models \phi(X)$ and $\phi(X) \wedge T(X, X') \models \phi(X')$.

In the following, we will use the notion of *case-defined functions* defined in a theory $\mathcal{T}$. These functions are defined by a sequence of couples $\{(case_i, value_i)\}_{i=1}^{n}$ where each $case_i$ is a predicate and each $value_i$ is a term (they correspond to statements of the form *if case_1 then value_1, else ...*); moreover, all the case predicates are required to be mutually exclusive, and their disjunction is a valid formula. Although case-defined functions are not standard in the SMT setting, they can be easily handled by using, for example appropriate if-then-else terms.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_20

B. Abstraction

Let C be $(X, I(X), T(X, X'))$ and $\tilde{C}$ be $(\tilde{X}, \tilde{I}(\tilde{X}), \tilde{T}(\tilde{X}, \tilde{X}'))$. Let S be the set of states of C, and $\tilde{S}$ be the set of states of $\tilde{C}$. Let α be a relation between S and $\tilde{S}$; we write $\alpha(s, \tilde{s})$ to denote that two states s and $\tilde{s}$ are in relation.

Definition 1. *We say that $\tilde{C}$ α-simulates C (written $C \rightarrow_\alpha \tilde{C}$) if the following two conditions hold:*

i. *For each initial state s of C, there exists an initial state $\tilde{s}$ of $\tilde{C}$ such that $\alpha(s, \tilde{s})$ holds.*
ii. *For each pair $(s, \tilde{s})$ such that $\alpha(s, \tilde{s})$ holds, and for each $s' \in S$ such that $s, s' \models T(X, X')$, there exists a state $\tilde{s}'$ such that $\alpha(s', \tilde{s}')$ holds and $\tilde{s}, \tilde{s}' \models \tilde{T}(\tilde{X}, \tilde{X}')$.*

If α is clear in the context, we might say that $\tilde{C}$ simulates (or *abstracts*) C.

Let $V \subseteq X \cap \tilde{X}$ be a set of variables, and $F(V)$ be a formula. We have the following facts about simulations:

Definition 2. *We say that the simulation $C \rightarrow_\alpha \tilde{C}$ preserves the formula F if, for all states s such that $s \models \neg F$, then for all $\tilde{s}$ such that $\alpha(s, \tilde{s})$, $\tilde{s} \models \neg F$.*

Proposition 1. *Given a simulation $C \rightarrow_\alpha \tilde{C}$ that preserves F, $\tilde{C} \models F \Rightarrow C \models F$.*

Proposition 2. *Given a simulation $C \rightarrow_\alpha \tilde{C}$ that preserves F, if F is inductive for $\tilde{C}$, then F is inductive for C.*

In many cases, the abstract variables $\tilde{X}$ of the system $\tilde{C}$ are different from the original variables X. In this paper, we consider only the case $\tilde{X} \subseteq X$ for the sake of simplicity.

If a counterexample is found in $\tilde{C}$, in general, this does not imply the existence of a counterexample in C. We say that a counterexample π in $\tilde{C}$ is spurious if there exists no path $s_0, \ldots, s_k$ in C such that $s_n \models \neg F$ and, for all $0 \leq i \leq k$, $\alpha(s_i, \pi[i])$. In such cases, the abstraction yields no helpful information and needs to be refined.

C. Asynchronous Composition

Let $C_1 = (X_1, I_1(X_1), T(X_1, X_1'))$ and $C_2 = (X_2, I(X_2), T(X_2, X_2'))$ be two symbolic transition systems. If V is a set of variables, we denote with $Inertia(V)$ the formula $\bigwedge_{v \in V}(v = v')$.

Definition 3. *The asynchrounous product between C_1 and C_2, is the transition system $C_1 \parallel C_2 = \big(X_1 \cup X_2, I_{C_1\parallel C_2}(X_1, X_2), T_{C_1\parallel C_2}(X_1, X_2, X_1', X_2')\big)$, where:*

- *$I_{C_1\parallel C_2}(X_1, X_2)$ is the formula $I_1(X_1) \wedge I_2(X_2)$;*
- *$T_{C_1\parallel C_2}$ is the formula $(T_1(X_1, X_1') \wedge Inertia(X_2 \setminus X_1)) \vee (T_2(X_2, X_2') \wedge Inertia(X_1 \setminus X_2))$.*

Given a set of variables $V \subseteq X_1 \cup X_2$ and a formula $F(V)$, asynchronous verification amounts to prove or disprove if $(C_1 \parallel C_2) \models F(V)$. More generally, if $V \not\subseteq X_i$, we may write $C_i \models F(V)$ with the meaning that we add to the X_i the remaining $V \setminus X_i$ variables, and we modify T_i by adding inertia on $V \setminus X_i$. We have:

Proposition 3. *If a formula F is not inductive for $C_1 \parallel C_2$, then there exists an $i \in \{1, 2\}$ such that F is not inductive for C_i.*

We say that two transition system C_1 and C_2 are *compatible* if each partial assignment to the shared variables can be extended to an initial state of C_1 if and only if it can be extended to an initial state of C_2. In practice, this means that the shared variables are initialized in the same way in the two systems.

Proposition 4. *Consider $C_1 \rightarrow_{\alpha_1} \tilde{C}_1$ and $C_2 \rightarrow_{\alpha_2} \tilde{C}_2$ two simulation relations. Suppose that $\tilde{C}_1$ and $\tilde{C}_2$ are compatible. Consider $\alpha_1 \parallel \alpha_2$ (called the product simulation) defined as*

$$\alpha_1 \parallel \alpha_2(s, \tilde{s}) \text{ iff } \alpha_1(s|_{X_1}, \tilde{s}|_{X_1}) \text{ and } \alpha_2(s|_{X_2}, \tilde{s}|_{X_2}).$$

Then, $C_1 \parallel C_2 \rightarrow_{\alpha_1\parallel\alpha_2} \tilde{C}_1 \parallel \tilde{C}_2$.

By definition of product simulation, we have the following corollary:

Corollary 1. *Consider $C_1 \rightarrow_{\alpha_1} \tilde{C}_1$ and $C_2 \rightarrow_{\alpha_2} \tilde{C}_2$ two simulation relation such that they both preserve a formula F. Assume that $\tilde{C}_1$ and $\tilde{C}_2$ are compatible. Then, $\alpha_1 \parallel \alpha_2$ also preserves F.*

III. A compositional approach with Abstraction Refinement

In this section, we outline a procedural framework that remains parametric, considering a generic family of transition systems, a generic abstraction procedure, and a target invariant property denoted as F. In the next section, we delve into a case study where we provide a more concrete setting.

Suppose that we have a finite family of transition systems $\{C_i\}_{i\in I}$. Let $C = \parallel_{i\in I} C_i$ be the asynchronous composition of the systems, and consider a formula $F(V)$ with $V \subseteq \bigcup_{i\in I} X_i$. The problem that we face is to prove or disprove whether $C \models F$. The problem is solved if either we find a counterexample, i.e. a path π of finite length n, such that $\pi[n] \models \neg F$, or if we find an inductive invariant Ψ for F. If F is not inductive itself, then by consecutive applications of Proposition 3 it follows that there exists a subset J of I such that F is not inductive for $\parallel_{j\in J} C_j$.

To describe our algorithm, we assume to have some subprocedures, namely: (i) a model checker, capable of automatically proving if an invariant holds in a transition system. If so, the model checker provides an inductive invariant for it. Otherwise, the model checker find a counterexample; (ii) a theorem prover, capable of checking whether a formula is inductive for a transition system, or if a counterexample can be simulated (e.g by bounded model checking). As a preprocessing step, we suppose to identify the set of components J for which the property is not already inductive. Moreover, let $\tilde{C}$ be a transition system such that there exists a simulation $\parallel_{j\in J} C_j \rightarrow \tilde{C}$. The only property that we require on the abstraction is that the simulation should preserve all inductive invariants found by the model checker. We consider the following procedure, depicted in Figure 1:

- we start by asking a model checker if $\tilde{C} \models F$. The model checker can either find an inductive invariant, Ψ, or a counterexample, π;
- If an invariant is found, we ask the prover to check if Ψ is also inductive for the whole asynchronous composition C. Note that, since the simulation preserves Ψ, we already know that it is inductive for the components $\{C_j\}_{j\in J}$ by Proposition 2.
- If the prover proves the induction, then we are done. Otherwise, there must exist a new set of components $J' \subseteq I \setminus J$ for which the induction check fails. We thus update the set J to be equal to $J \cup J'$, and we restart the loop by updating the abstraction $\tilde{C}$.
- Suppose instead that the model checker finds a counterexample in $\tilde{C}$. Then, we ask the prover if the counterexample can be simulated by C. If so, the algorithm terminates with a counterexample. Otherwise, we refine the abstraction to remove the abstract counterexample.

The key differences of our approach from conventional CEGAR methods in compositional verification such as [13], [7] lies in the fact that the system $\tilde{C}$ doesn't abstract entire composition $C =\|_{i\in I} C_i$; instead, it abstracts only the under-approximation $\|_{j\in J} C_j$. We could eventually abstract all components, when $J = I$, and $C \to \tilde{C}$ is simulation - but our method is best suited for situations where this should not happen.

Even when $\tilde{C}$ doesn't represent the entire system, the algorithm's soundness is evident. This is because we conduct an additional induction check to verify whether the invariant identified during model checking is also inductive for the broader composition C.

IV. Verification of concurrent parameterized systems

In this section, we illustrate the application of the procedure outlined in Section III through a specific use case. Our algorithm was conceived to facilitate the verification of interlocking logic within railway stations, as part of a larger initiative to integrate various formal methods into railway design [1]. In this scenario, the system's components are each represented by parameterized transition systems. These components interact by sharing multiple variables; our objective is to ascertain the general safety of the composition of them. Despite the presence of a vast number of components, only a subset is critical for ensuring safety. Furthermore, while the systems may involve a large number of variables, we recognize that only a limited number are pertinent to safety concerns. Consequently, our approach to abstraction focuses on narrowing down to these essential variables.

The systems are modeled with a subclass of the formalism of [14] and similar to [9], where parameterized systems are modeled as symbolic transition systems with state variables that are functions from an uninterpreted theory (with finite but unbounded universes) to a generic theory. Moreover, quantifiers occur in the system description to model the unboundedness of possible instances.

Definition 4. *An array-based transition system $C = (X, I(X), T(X, X'))$ is a symbolic transition system where:*

- *X are a set function symbols;*
- *$I(X)$ is a formula of the form $\forall j. \bigwedge_{x\in Y} x(j) = val_x$ - where val_x is a constant of the appropriate signature, and $Y \subseteq X$;*
- *$T(X, X')$ is a disjunction of formulae of the form (called transition rules)*

$$\exists i.(\phi_G(i, X) \wedge\ \phi_U(i, X, X')) \tag{1}$$

where $\phi_G(i, X)$ is called the guard, and $\phi_U(i, X, X')$ is the functional update, i.e., a formula of the form

$$\forall j. \bigwedge_{x\in X} x'(j) = F_x(i, j, X, X')$$

with $\{F_x\}_{x\in X}$ a family of case-defined function.

We start by defining the simulation relation that we will use.

Definition 5. *Let $V \subseteq X$ a set of variables. We define a relation α between two assignments $s, \tilde{s}$ of X and V by*

$$\alpha(s, \tilde{s}) \Leftrightarrow s|_V \equiv \tilde{s},$$

i.e. we ask that the two states are in relation iff they assign the same value to the variables in V.

Given a (sub)set of variables V and a transition system C defined as in Definition 4, we now define a new transition system $\tilde{C}^V$ such that there exists a simulation between the two systems. The new variables will be $V \cup B \cup E$, with B and E sets of fresh input variables. The abstract initial formula of the system, denoted $\tilde{I}(V)$, is simply obtained from I by dropping the conjuncts that are not assigning variables in V; that is, we have that

$$\tilde{I}(V) = \bigwedge_{v\in V} \forall j.v(j) = val_v.$$

For the abstract transition formula, denoted as $\tilde{T}(V, B, E, V')$, we need more steps. We will work on the single transition rules of the concrete transition, that are of the form (1). The abstract transition will be a disjunction of formulae of the form $\exists i.(\tilde{\phi_G}(i, V) \wedge\ \tilde{\phi_U}(i, V, B, E, V'))$ where

- $\tilde{\phi_G}(i, V)$ is obtained by ϕ_G by replacing each atom that contains variables in $X \setminus V$ with the constant $true$, if it occurred positively in the formula, or $false$ otherwise;
- $\tilde{\phi_U}(i, V, B, E, V')$ is the formula $\bigwedge_{v\in V} \forall j.v'(j) = \tilde{F_v}(i, j, V, B, E, V')$ where $\tilde{F_v}$ is a case-define function with a sequence of $\tilde{case}_i(V, B)$ statements and a sequence of corresponding terms $\tilde{val}_i(V, B, E)$ such that:
  - $\tilde{case}_i(V, B)$ is either equal to $case_i(V)$ if the original case predicate is defined only over the V variables, or is a fresh boolean constant $b \in B$ otherwise;
  - $\tilde{val}_i(V, B, E)$ is either equal to $val_i(V)$ if the original term is defined only over V, or is a fresh constant $e \in E$ of the appropriate type otherwise.

Let $\tilde{C}^V = (\{V, B, E\}, \tilde{I}(V), \tilde{T}(V, B, E, V'))$. We have:

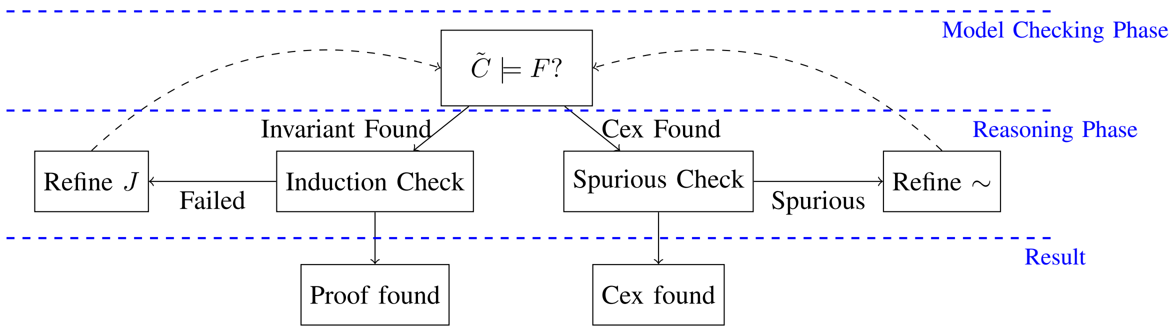


Fig. 1. The procedure

Proposition 5. *$\tilde{C}^V$ simulates C.*

Moreover, since the simulation relation is the equality on V, we have that:

Proposition 6. *The simulation preserves each formula $F(V)$ defined only over the set of variables V.*

Therefore, in order for the abstraction $\sim^V$ to preserve the property, we need that the set of variables V always include all the state variables occuring in the property to prove. Thus, from Corollary 1, we have that, given any family of *compatible* array-based transition systems:

Corollary 2. *$\| \; C_i \; \rightarrow \| \; \tilde{C}_i^V$ via the product simulation. Moreover, the product simulation preserves all the formulas defined over V.*

Thanks to the latter, we can use as an abstraction for $\|_{i \in J} C_i$ the composition of the individual abstractions, i.e. $\|_{i \in J} \tilde{C}_i^V$

A. Refinement

Suppose that, during the model checking phase, we found an abstract counterexample π. For refinement, we check as usual the satisfiability of the concrete unrolling. In the case of satisfiability, we are in the presence of a real counterexample, and we can exit from the procedure. In case of unsatisfiability, we are in presence of a spurious counterexample, and we follow this refinement procedure: we start by computing an unsat core of the latter formula. Then, let V' be the set of variables that occur in at least one literal of the core and not in V. We update V to be $V \cup V'$. We have the following result that ensures that ensures that V' is never empty:

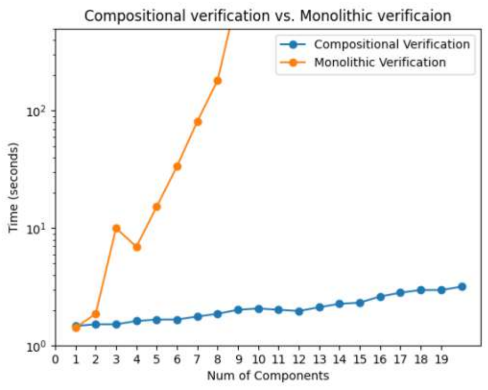
Proposition 7. *In case a spurious counterexample in $\|_{i \in J} \tilde{C}_i^V$ is found, then there exists a literal in the unsat core of the concrete unrolling that contains a variable not occurring in V.*

This refinement allows us to have a notion of progress, since at each spurious counterexample we decrease the number of variables that not abstracted.

B. Implementation and first results

We developed the algorithm presented in the preceding section using Python3, leveraging the SMT solvers Z3 and Mathsat, along with the parameterized model checker Lambda [4]. Given that Mathsat lacks support for quantified formulae, we exclusively utilized Z3 for the 'Induction Check' sub-procedure. For 'Spurious Check' and 'Refinement' sub-procedures, both Mathsat5 and Z3 were employed. Lambda, capable of processing system descriptions in the VMT language [6], is able to synthesize inductive invariants for systems as defined in Definition 4.

We tested the algorithm on a simplified case study of the railway logic, with 5 parameterized components (with a total of 15 variables) and two properties. For the set of abstracted variables V, we always initialize it with the set of state variables occuring in the property to prove. The first property was verified by the algorithm in 4.2 seconds, by using only 5 variables and 2 components, and one refinement step. Instead, the monolithic approach (i.e. using the model checker on the entire system description of the composition) took around 9 seconds. On the other hand, the second property was a false assertion whose counterexample involved most of the components and variables used. In that case, the monolithic approach can find a counterexample faster then the compositional algorithm. Although these outcomes are not entirely satisfactory, it's crucial to acknowledge that our case study was quite limited compared to the actual system. In reality, the system comprises over 100 components, each with numerous variables, and a full symbolic description has yet to be achieved. A second source of benchmarks derives from two parameterized protocols, which we have adapted by integrating N components capable of altering certain shared variables. We test the algorithm on properties that are true and are independent of the modifications enacted by these additional components. The results are depicted in Figure 2. The x-axis represents the number of components, while the y-axis measures the time taken by the procedures in seconds (presented on a logarithmic scale). Compared to the monolithic approach (in orange), our algorithm's verification process (in blue) is significantly less time-consuming and remains relatively constant. A virtual machine to replicate the results is available here, together with an extended version of this paper.



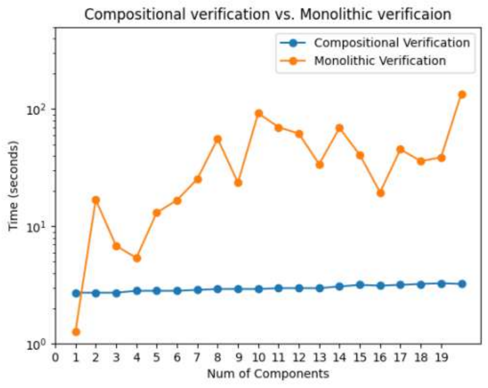


Fig. 2. Results on protocols with additional components

V. Conclusions and future work

In this paper, we introduced a method for verifying the safety properties of asynchronous compositions of symbolic transition systems defined over an SMT theory $\mathcal{T}$. We believe that this method is particularly suited for scenarios requiring the verification of properties across a large family of components, where the inductive invariant can be identified by examining a subset of those components. If the procedure terminates by abstracting only a subset of the components, then we can determine *a posteriori* that a split invariant [10] between the abstracted and non-abstracted components is found.

We applied this general algorithm to a family of symbolic transition systems designed to describe parameterized systems and defined a form of splicing as an abstraction strategy that concentrates on a subset of the system's variables. A prototype of the algorithm was developed and tested on simple benchmarks. The initial results are promising, and we plan to extend its application to a comprehensive verification project focused on interlocking logic, where the previously described situation (numerous components with many variables) frequently arises. Moreover, future work may integrate assumption-guarantee methods in our approach, such as those found in [2], [12], [11], [8] to further simplify the verification.

References

[1] Cavada, R., Cimatti, A., Griggio, A., and Susi, A. A formal IDE for railways: Research challenges. In *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops - AI4EA, F-IDE, CoSim-CPS, CIFMA, Berlin, Germany, September 26-30, 2022, Revised Selected Papers* (2022), P. Masci, C. Bernardeschi, P. Graziani, M. Koddenbrock, and M. Palmieri, Eds., vol. 13765 of *Lecture Notes in Computer Science*, Springer, pp. 107–115.

[2] Cimatti, A., Dorigatti, M., and Tonetta, S. OCRA: A tool for checking the refinement of temporal contracts. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013* (2013), E. Denney, T. Bultan, and A. Zeller, Eds., IEEE, pp. 702–705.

[3] Cimatti, A., Griggio, A., Mover, S., Roveri, M., and Tonetta, S. Verification modulo theories. *Formal Methods Syst. Des. 60*, 3 (2022), 452–481.

[4] Cimatti, A., Griggio, A., and Redondi, G. Verification of SMT systems with quantifiers. In *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings* (2022), A. Bouajjani, L. Holík, and Z. Wu, Eds., vol. 13505 of *Lecture Notes in Computer Science*, Springer, pp. 154–170.

[5] Cimatti, A., Griggio, A., and Redondi, G. Towards the verification of a generic interlocking logic: Dafny meets parameterized model checking, 2024.

[6] Cimatti, A., Griggio, A., and Tonetta, S. The VMT-LIB language and tools. *CoRR abs/2109.12821* (2021).

[7] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *Computer Aided Verification* (Berlin, Heidelberg, 2000), E. A. Emerson and A. P. Sistla, Eds., Springer Berlin Heidelberg, pp. 154–169.

[8] Gheorghiu Bobaru, M., Păsăreanu, C. S., and Giannakopoulou, D. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification* (Berlin, Heidelberg, 2008), A. Gupta and S. Malik, Eds., Springer Berlin Heidelberg, pp. 135–148.

[9] Ghilardi, S., and Ranise, S. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Log. Methods Comput. Sci. 6*, 4 (2010).

[10] Giannakopoulou, D., Namjoshi, K. S., and Pasareanu, C. S. Compositional reasoning. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 345–383.

[11] Gupta, A., McMillan, K. L., and Fu, Z. Automated assumption generation for compositional verification. In *Computer Aided Verification* (Berlin, Heidelberg, 2007), W. Damm and H. Hermanns, Eds., Springer Berlin Heidelberg, pp. 420–432.

[12] Limbrée, C., Cappart, Q., Pecheur, C., and Tonetta, S. Verification of railway interlocking - compositional approach with ocra. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification* (Cham, 2016), T. Lecomte, R. Pinger, and A. Romanovsky, Eds., Springer International Publishing, pp. 134–149.

[13] Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. Property preserving abstractions for the verification of concurrent systems. *Form. Methods Syst. Des. 6*, 1 (jan 1995), 11–44.

[14] Redondi, G., Cimatti, A., Griggio, A., and McMillan, K. Invariant checking for smt-based systems with quantifiers. *ACM Trans. Comput. Logic* (aug 2024). Just Accepted.

Semi-open-state testing for in-silicon coherent interconnects

Jasmin Schult, Ben Fiedler, David Cock, Timothy Roscoe
ETH Zürich, Zürich, Switzerland
firstname.lastname@inf.ethz.ch

***Abstract*—In this paper, we extend open-state conformance testing from Mealy FSM specifications to implementations where only a subset of states are observable. We show that the classical transition tour can be used to completely test such implementations for conformance, including unobservable states, under the assumption that in the specification, any trace from an unobservable state eventually reaches an observable state – i.e. that the observable states form a feedback vertex set. Complete transition tour testing can efficiently test for many conformance relations when they are appropriately formulated. Generalized-quasi-reduction (GQR) is useful for protocol testing because it allows for partial, non-deterministic specifications, but establishing GQR in the general setting is complex and expensive. We show a relation that implies GQR and is practical for transition tour testing.**

Our setting of partial state observability applies to an important class of protocol implementations in modern hardware: cache coherence. These protocols are nearly universal in multiprocessor systems, and are notoriously difficult to verify both at the specification and implementation level. We show that their structure lends them naturally to complete open-state testing under this extended definition. During design, coherence protocols are elaborated from an FSM of *stable states* with atomic, observable transitions by the addition of a large number of unobservable *transient states* to handle concurrency, including out-of-order and interleaved message delivery. We demonstrate that a real in-silicon implementation on the Cavium ThunderX-1 CN88XX CPU has exactly the required characteristics and we establish the GQR conformance relation against a specification of its inter-socket coherence protocol.

I. Introduction

In this paper, we introduce *semi-open-state testing*, a test setting where an implementation with partially visible states is tested from a finite state machine model of its behavior to establish a conformance property like GQR. We show that under a set of additional requirements that we impose on semi-open-state testing, complete visibility of the implementation's states is achieved, thus allowing simpler *open-state testing* techniques [1] to be employed instead of the more expensive, general complete Finite State Machine (FSM) testing methods [2]. This work arises out of our analysis of *cache coherence protocols* in the context of the Enzian [3] project. We show that this practically-important class of protocols match the requirements of semi-open-state testing by demonstrating a complete, online validation that the native protocol of the server-class CN88XX (ThunderX-1) CPU is GQR-conformant to its specification. This testing is conducted live against the actual silicon implementation.

Semi-open-state testing is motivated by the structure of hardware cache coherence protocols, particularly in emerging standards such as CXL [4]. These protocols consist of a set of *stable* states that are generally visible via debug mechanisms, plus a larger number of intermediate *transient* states added to handle the effects of concurrency between nodes e.g. message reordering. A typical protocol has 3–5 stable states, and 10–100 transient states. The degree of reordering is bounded in practice, and thus a transient state will lead back to a stable state after a finite number of messages. Our insight is to label these intermediate states with the last stable state and this finite, observable IO trace, thereby reducing the task to standard open-state testing. Open-state testing methods are both simpler and less expensive, which greatly increases their practicality for testing real hardware.

Formally, this work concerns the *complete conformance testing* [5] of implementations against *Mealy FSM* specifications. Existing work leverages either general complete FSM testing or *open-state testing*. The general complete testing methods only assume an upper bound m on the number of states in the implementation, hence they are referred to as m-*complete* methods. They are comparatively heavy-weight and worst-case exponential in m. Open-state testing, on the other hand, derives large gains in efficiency and implementation complexity from its stronger assumptions on the system under test. Chief among these is that all implementation states are *visible*. The first challenge in applying open-state techniques to coherence protocols is thus the invisibility of transient states. We overcome this by developing semi-open-state testing, the first key contribution of this paper.

The second challenge is in identifying a conformance relation that matches the characteristics of a coherence protocol. We settle on GQR principally for its ability to permit partial and non-deterministic specifications, which is essential in a practical protocol specification to leave sufficient freedom for implementors to optimize their designs. GQR is presented in an m-complete setting, and we know of no existing open-state-compatible formulation in the literature. Our second contribution is thus the open-state-testable relation GQR_{open}, which we prove is testable under the assumptions of semi-open-state testing and sufficient to establish standard GQR for the systems we consider.

In section II, we introduce both conformance testing from Mealy FSMs and GQR as used in existing literature. We develop semi-open-state testing in section III, enumerating

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_21

the assumptions needed to extend open-state testing to semi-visible implementations. In section IV we explain the function and characteristics of cache coherence protocols, why semi-open-state testing is applicable to them and why the emergence of coherent interconnect standards necessitates such testing. We demonstrate how this approach performs in practice in section V, by applying semi-open-state testing to the cache coherence implementation of the ThunderX-1, which is deployed as part of the Enzian platform. Finally, we conclude and outline future work in section VI.

II. Background and related work

A. Notation & Definitions

We consider Mealy-type IO state machines whose output depends both on state and current input, and adapt the notation of Hierons [6]. A machine is a tuple $M = (S, s_0, X, Y, h)$ where:

- S is the finite set of states.
- $s_0 \in S$ is the initial state.
- X is the finite input alphabet.
- Y is the finite output alphabet.
- $h \subseteq (S \times X) \times (S \times Y)$ is the transition relation.

Where necessary, a subscript identifies the associated machine, e.g. h_{SPEC} for the transition relation of machine SPEC.

If $((s,x),(s',y)) \in h$ then on receiving input x when in state s, the machine may transition to state s' while producing output y. $h(s,x)$ denotes the image of $\{(s,x)\}$ under h, i.e. the allowed transitions for a given state and input.

We assume that *all states in S are reachable* from the machine's initial state, since unreachable states are not relevant to the observable behavior of the machine.

We assume that M is *observably non-deterministic*: The target state is completely determined by the observed input and output. Equivalently, the outputs of any two distinct reactions in $h(s,x)$ must differ:

$$\{(s',y'),(s'',y'')\} \subseteq h(s,x) \implies s' = s'' \vee y' \neq y'' \quad (1)$$

We denote the set of inputs x for which some transition of machine M is defined from state s by

$$\Omega_M(s) = \{x \mid ((s,x),(\_,\_)) \in h_M\}$$

M is *completely specified* if $\forall s.\ \Omega_M(s) = X$ i.e. its behavior is specified for all inputs in all states.

Y contains the distinct special symbols $\sim$ and $\perp$ representing no output and failure. $\sim$ is allowed for both specification and implementation, but $\perp$ only for an implementation. $\sim$ in a specification requires that an implementation produce no output, and $\sim$ in an implementation is assumed to be observable (e.g. by timeout [7]). $\perp$ does not satisfy any specification. With these additions we may assume that any implementation is completely specified.

Every machine step consumes exactly one input, and produces exactly one output. A length-n trace of the machine records both state transitions and the corresponding input/output pairs:

$$((s,x_1),(s_1,y_1)),\cdots,((s_{n-1},x_n),(s_n,y_n))$$

The corresponding *IO trace* is the projection containing only the message pairs:

$$x_1/y_1,\cdots,x_n/y_n$$

We write $t.x/y$ for the IO trace t extended with the IO transition x/y and t_1t_2 for trace concatenation.

The language $L_M(s)$ is the set of permitted IO traces beginning with machine M in state s, and for any IO trace $t \in L_M(s)$ of an observably non-deterministic machine,

$$\text{AFTER}_M(s,t)$$

is the unique final state s' that machine M reaches after observing trace t beginning in state s. We omit the state s if it corresponds to the initial state: $L_M := L_M(s_0)$ and $\text{AFTER}_M(t) := \text{AFTER}_M(s_0,t)$.

Note that

$$\text{AFTER}_M(s,(tt')) = \text{AFTER}_M((\text{AFTER}_M(s,t)),t') \quad (2)$$

B. Conformance Testing against Mealy Machines

Testing an implementation against a specification is a well-studied problem, known as conformance testing, model-based testing, and fault detection in the literature. Different techniques exist to test against a variety of formal models; This paper considers conformance testing against Mealy FSMs.

Establishing a relation between a real-world implementation and an abstract specification SPEC begins with the *testability hypothesis* [8]. For Mealy FSM specifications, this includes that the behavior of the implementation can be captured by an unknown, abstract Mealy FSM denoted by IMPL.

For IMPL to be testable even if its behavior is non-deterministic requires us to assume (at least) *weak fairness*: If the implementation is presented with input x in state s enough times, it will eventually take every possible transition (to some state s' with output y).

Existing work [5]–[7], generally considers IMPL to be a *completely specified*, *observably non-deterministic* Mealy FSM. Complete specification captures the practical reality that a real-world implementation cannot refuse an input provided by its environment and so always defines some reaction, even if this is just failing or producing no output. These are the $\sim$ and $\perp$ output symbols already introduced. Further, any completely-specified Mealy FSM can be transformed into an observably non-deterministic equivalent via the standard NFA–DFA construction [9].

Requirement 1. *(Testability Hypothesis) In practice, the testability hypothesis requires a suitable* I/O adaptor *to translate between the implementation's concrete and the specification's abstract I/O. Aarts et al. formalize the requirements of such an adaptor in the context of state machine learning [10]. Furthermore, the abstract implementation needs to be* input-driven *by the input alphabet of* SPEC.

For the fairness assumption to hold in practice, it is sufficient that the implementation's non-determinism originate only *from true randomization. In particular, the implementation's non-determinism* must not *originate from concurrent* *processing of inputs resulting in different outcomes (which would require testing to generate unknown relative input timings to explore all such outputs). Equivalently, the implementation's processing of inputs must be* linearizable.

It is important to ascertain that a real-world implementation exhibits these properties before testing it against a Mealy specification. We revisit these assumptions in the context of coherence protocols in section IV.

Testing seeks to establish a *conformance relation* between SPEC and IMPL. In this case we work with GQR as defined by Hierons [6]. GQR allows partial specifications, i.e. it does not require a transition in response to every input in every state. This is necessary for protocols, which are typically sparse; only a small set of inputs are expected in a particular state. GQR interprets non-determinism in the specification as implementation choice. An implementation is a *generalized quasi reduction* of its specification if all its outputs are valid choices offered by the specification, so long as only inputs defined by the specification are provided to it. More formally,

Definition 1 (Generalized-Quasi-Reduction (GQR))**.**

$$\forall t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}, x \in \Omega_{\text{SPEC}}(\text{AFTER}_{\text{SPEC}}(t)). \\ \{y | t.x/y \in L_{\text{IMPL}}\} \subseteq \{y | t.x/y \in L_{\text{SPEC}}\} \quad (3)$$

This definition simplifies the original in two ways: Firstly, as our SPEC is observably nondeterministic, $\text{AFTER}_{\text{SPEC}}(t)$ is unique, and thus we need only quantify over its inputs and not over the set of possible states as well. Secondly, as IMPL is completely specified we can drop the first condition, which requires a corresponding implementation transition.

Conformance testing methods are either *complete* or *incomplete*. Complete methods can formally establish a conformance relation, whereas incomplete methods increase the confidence that a relation holds, but without formal guarantees. We consider only complete conformance testing.

The difficulty of complete conformance testing depends on the additional assumptions made on the implementation. Without any assumptions, the IO behavior of even a finite machine may be infinite, and thus impossible to completely test with finite methods.

Complete testing generally assumes a known upper bound m on the number of implementation states, giving *m-complete testing* [6]. The number of transitions that must be tested is exponential in m, as shown by Moore [11]. In particular, it is insufficient to cover all transitions of the specification once [2]. Note that this is true even if the implementation is assumed to have at most as many states as the implementation, since the implementation may enter the wrong state after a transition. This is due to the implementation state not being visible: it can only be inferred from the future I/O traces of the implementation.

If the state is visible, the complexity decreases dramatically. This is *open-state testing* [1] or testing with a reliable *status message* [5], [12]. The finite representations of specification and implementation can be compared directly, instead of reasoning over all their (potentially) infinitely many I/O traces. Complete testing against a specification reduces to an online traversal of its transition graph, with sufficient repetitions to account for non-determinism. Bourdonov et al. [1] describe a traversal algorithm that can cope with detours caused by unfavorable choices of non-determinism in the implementation.

Open-state testing is not only simpler to implement but also more efficient than m-complete testing. Sidhu et al. [13] show that transition traversal generates shorter test cases on average compared to m-complete methods for deterministic, complete specifications; this also holds even if the bound m is equal to the number of states in the specification. Open-state testing is much more efficient than the general methods used to establish GQR, as testing against partial, non-deterministic specifications is considerably more difficult [6], [14]. Furthermore, open-state testing is able to detect if an implementation possesses any number of additional states with respect to the specification without an increase in testing complexity, whereas we pay an exponential tax to do the same for m-complete methods.

In the following sections, we show that m-complete testing for GQR can be reduced to open-state testing, even if only a subset of states are actually visible (section III), and demonstrate that the necessary assumptions apply to an important class of practical protocols: those for hardware cache coherence (section IV).

III. Semi-open-state Testing and GQR_{OPEN}

Requirement 2. *(Visible States) We require that a finite subset of reachable implementation states, including the initial state, are known a priori:*

$$s0_{\text{IMPL}} \in \text{S}_{known} \subseteq S_{\text{IMPL}}$$

These states are fully observable—we see whether or not the implementation is in such a state and if so, which one. All others are only indirectly observable by their IO behavior.

We thus obtain a partial labelling function, λ, from traces to states:

$$\text{AFTER}_{\text{IMPL}}(t) = s \wedge s \in \text{S}_{\text{known}} \implies \lambda(t) = s \quad (4)$$

We label the unknown states by reference to the last known state. For any trace t from s_0, take the longest t_K such that

$$t_K t_U = t \qquad \lambda(t_K) \in \text{S}_{\text{known}}$$

We thus obtain a complete labelling of traces, $\hat{\lambda}$, with the tuple of last known state and IO trace since that state:

$$\hat{\lambda}(t) = (\lambda(t_K), t_U) \quad (5)$$

As IMPL is observably nondeterministic, the trace uniquely defines the final state, which is thus observable given the observed label of the IO trace to the last known state (t_K), plus the IO trace through all subsequent unknown states (t_U).

For any known state s_K, a corresponding trace t_K, and a continuation t_U such that $t := t_K t_U \in L_M$ (or equivalently

$\hat{\lambda}(t) = (s_K, t_U)$), the state s corresponding to trace t's label matches the machine's actual state after t:

$$
\begin{aligned}
s &\coloneqq \text{AFTER}_{\text{IMPL}}(\hat{\lambda}(t)) \\
&= \text{AFTER}_{\text{IMPL}}(\lambda(t_K), t_U) && \text{by (5)} \\
&= \text{AFTER}_{\text{IMPL}}(\text{AFTER}_{\text{IMPL}}(t_K), t_U) && \text{by (4)} \\
&= \text{AFTER}_{\text{IMPL}}(t_K t_U) && \text{by (2)} \\
&= \text{AFTER}_{\text{IMPL}}(t) && (6)
\end{aligned}
$$

Hence $\hat{\lambda}$ is equivalent to labelling states with the full trace from s_0 which, by observable nondeterminism, uniquely identifies the state. $\hat{\lambda}$ thus allows a tester to identify the implementation state after any trace, providing the complete visibility of the implementation's states needed for open-state testing.

To complete the reduction to open-state testing, the states of SPEC need to match the observations returned by $\hat{\lambda}$. To preserve the finiteness of S_{SPEC}, we need to impose the following requirement on the specification that we wish to test against:

Requirement 3. *(Feedback Vertex Set) We require that the* known states *form a feedback vertex set*[1] *in the specification's transition graph, and every trace through only unknown states is finite. We further require that there exists some global bound, c, on the length of all such traces.*

The transformation of the original specification to a *semi-open testable* SPEC entails converting the DAGs between the stable states in its transition graph to trees by duplicating states as necessary. This transformation may therefore increase the number of transitions and states to $O(|S_{known}| \cdot (|X| \cdot |Y|)^c)$ in the worst case, depending on the shape of the transition graph. We therefore need to carefully examine the specification to determine if open-state testing remains practical and preferable to m-complete testing:

Requirement 4. *(Practicability) The bound c and the shape of the original specification should be carefully examined to determine if the size of the corresponding* semi-open testable SPEC *remains manageable.*

With the resulting transformed SPEC, our reduction of semi-open-state testing to open-state testing is complete.

We now shift our attention to how open-state testing can be leveraged to establish GQR. A relation is *open-state testable* if, assuming that the tester can drive the machine's input and observe both its state and output, it is possible to determine if a specification and implementation lie in the relation by driving the implementation through all input transitions in the specification a finite number of times (to account for non-determinism). This implies that an *open-state testable* relation can be established by a graph traversal of the specification, as in the algorithm of Bourdonov [1].

GQR as formulated in Definition 1 is not directly *open-state testable*. It permits a single specification state to be implemented as multiple states that are distinguishable only by their inconsistent choices of non-deterministic options. For a detailed discussion of this phenomenon, we refer to the study of the classical reduction relation by Petrenko et al. [14]. In the open-state setting, the observations of these multiple implementation states will fail to match the single state in the specification, causing the test to fail even if the implementation is GQR in the general sense.

We therefore propose the following *open-state-testable* specialization of Definition 1, which assumes the observability of implementation states:

Definition 2 (Open-State GQR (GQR_{open})).

$$
\begin{aligned}
& s0_{\text{IMPL}} = s0_{\text{SPEC}} \land \\
& \qquad \forall s \in S_{\text{SPEC}}.\ \forall x \in \Omega_{\text{SPEC}}(s). \\
& \qquad \big(\exists t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}.\ s = \text{AFTER}_{\text{IMPL}}(t)\big) \\
& \qquad\qquad \implies h_{\text{IMPL}}(s,x) \subseteq h_{\text{SPEC}}(s,x) \quad (7)
\end{aligned}
$$

Notice that GQR_{open} is comparing states of the implementation and states of the specification for equality (in particular, recall that the transition functions h_{SPEC} and h_{IMPL} map to sets of (next state, output)). This is possible because the states that the implementation adopts are completely visible in the open state setting and can hence act like an additional output of the implementation. Recall that for the semi-open-state testing setting that we have discussed earlier, this visibility is provided by the $\hat{\lambda}$ function.

GQR_{open} implies, inductively, that the states of the implementation that are reached by traces defined by the specification must agree with those of the specification:

Lemma 1.

$$
\text{GQR}_{open} \implies \forall t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}.\quad \text{AFTER}_{\text{IMPL}}(t) = \text{AFTER}_{\text{SPEC}}(t)
$$

Initially,

$$
\text{AFTER}_{\text{IMPL}}([]) = s0_{\text{IMPL}} = s0_{\text{SPEC}} = \text{AFTER}_{\text{SPEC}}([])
$$

Take any

$$
t.x/y \in L_{\text{SPEC}} \cap L_{\text{IMPL}} \tag{8}
$$

such that

$$
s \coloneqq \text{AFTER}_{\text{IMPL}}(t) = \text{AFTER}_{\text{SPEC}}(t) \tag{9}
$$

Since $t.x/y \in L_{\text{SPEC}} \cap L_{\text{IMPL}}$, by the definition of h_M:

$$
\begin{aligned}
& x \in \Omega_{\text{SPEC}}(s) && (10) \\
& (\text{AFTER}_{\text{IMPL}}(t.x/y), y) \in h_{\text{IMPL}}(s,x) && (11) \\
& (\text{AFTER}_{\text{SPEC}}(t.x/y), y) \in h_{\text{SPEC}}(s,x) && (12)
\end{aligned}
$$

Given (8,9, and 10), GQR_{open} *(7) yields:*

$$
h_{\text{IMPL}}(s,x) \subseteq h_{\text{SPEC}}(s,x)
$$

or, the implementation's transitions are a subset of the specification's.

[1]A feedback vertex set of a directed graph G is a set of vertices whose removal transforms G into a directed acyclic graph.

Combined with (11) we have that

$$(\textsc{After}_{\text{IMPL}}(t.x/y), y) \in h_{\text{SPEC}}(s, x)$$

or, the final implementation state is among those of the specification with the same observable IO behaviour.

The actual specification state must also be in this set (12), and thus by the definition of observable non-determinism (1) must be equal to the implementation state:

$$\textsc{After}_{\text{IMPL}}(t.x/y) = \textsc{After}_{\text{SPEC}}(t.x/y)$$

□

Lemma 2. GQR$_{open}$ *is open-state testable.*

Definition 2 applies to the set of implementation states reachable by a trace also accepted by the specification. This set is finite. Assuming weak fairness (Requirement 1), we will eventually observe every possible y for each s and x. Thus by repeatedly traversing every input transition we will terminate, having exhaustively tested the subset relation. □

Lemma 3. GQR$_{open}$ $\implies$ GQR

Using Definition 1 (GQR) take $t \in L_{\text{IMPL}} \cap L_{\text{SPEC}}$, $x \in \Omega_{\text{SPEC}}(\textsc{After}_{\text{SPEC}}(t))$ *and* y *such that* $t.x/y \in L_{\text{IMPL}}$.

Since $t, t.x/y \in L_{\text{IMPL}}$,

$$\exists s'.\ (s', y) \in h_{\text{IMPL}}(\textsc{After}_{\text{IMPL}}(t), x) \qquad (13)$$

Moreover, from GQR$_{open}$ *we have, by Lemma 1*

$$\textsc{After}_{\text{SPEC}}(t) = \textsc{After}_{\text{IMPL}}(t)$$

Thus, by GQR$_{open}$ *(7)*

$$h_{\text{IMPL}}(\textsc{After}_{\text{IMPL}}(t), x) \subseteq\ h_{\text{SPEC}}(\textsc{After}_{\text{SPEC}}(t), x)$$

Together with (13) we have

$$(s', y) \in h_{\text{SPEC}}(\textsc{After}_{\text{SPEC}}(t), x)$$

Thus, since $t \in L_{\text{SPEC}}$*:*

$$t.x/y \in L_{\text{SPEC}}$$

□

In this section we have presented semi-open-state testing for semi-visible machines, and GQR$_{\text{open}}$: a sufficient, semi-open-state-testable condition for GQR to hold on the traces of such a machine. By exploiting the observability of the subset of known states and the observable nondeterminism of the implementation, we construct a complete labelling, $\hat{\lambda}$, of implementation states. Given an appropriately shaped specification, this allows us to reduce semi-open-state testing to open-state testing, avoiding the cost of general m-complete testing.

IV. Cache-coherence protocol interoperability

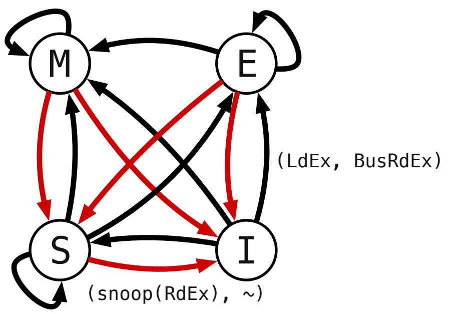
We now turn to a key real-world problem which is highly amenable to semi-open-state testing. Indeed, our motivation to develop the formalism stemmed from a practical problem we faced: how to gain confidence that two different endpoint implementations of an informally-defined and under-specified cache coherence protocol will successfully interoperate. The endpoints of mainstream inter-processor cache coherence protocols turn out to be an excellent match to the requirements for semi-open-state testing.

Modern computers with multiple processor cores rely on caches for performance: a hierarchy of caches holds copies of data from memory (*lines*), and these caches are kept *coherent* by a hardware *cache coherence protocol* which ensures that, at any point in time, all copies of a line that reside in the system's caches are identical [15]. This is a global invariant that must be upheld at all times; the protocol maintains this invariant while serving memory requests (reads and writes) made by different cores. To correctly and efficiently negotiate the simultaneous handling of multiple such requests, the endpoints of modern coherence protocols tend to be large and complex state machines. At the same time, high assurance in the correct operation of these endpoints is required: bugs prevent correct execution of the entire machine, and the performance-critical implementation in silicon means that these bugs can rarely be fixed post-silicon. For this reason, formal methods have long been employed in coherence protocol designs [16], e.g. for verifying the protocol definitions [17], or generating correct-by-construction protocol state machines [18].

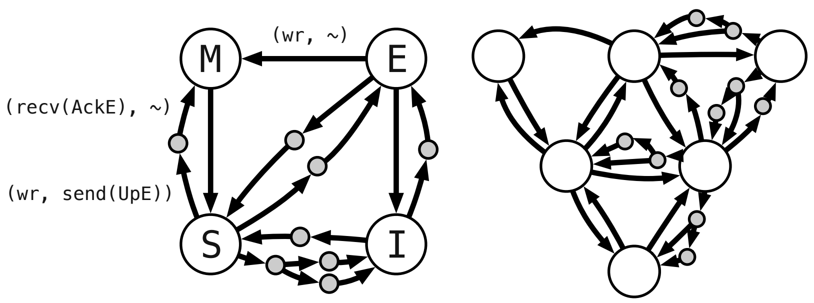
Work to date that tests if hardware implementations correctly implement these verified protocol designs has operated on the entire coherent system, rather than on individual protocol endpoints as we are proposing: Kahlouche et al. [19] and Kriouile et al. [20] also generate tests from formal models, but the scale of the system-wide protocol makes complete testing intractable and simulation environments are needed to exert control over the concurrent execution at the protocol endpoints. Consequently, these works have neither addressed the visibility of endpoint states nor leveraged those states to achieve complete testing coverage. Orthogonally, other efforts [21], [22] have integrated additional testing logic into the system implementation to generate test stimuli and to directly check the high-level protocol invariants.

These existing testing methods have worked so far because coherence protocols have generally been specific to a particular processor model, allowing a single hardware team to conduct the design, verification, and implementation of the entire coherent system.

This situation is changing: open, cache-coherent interconnect standards like CXL.cache [4], NVlink [23], CCIX [24], and TileLink [25] attach a range of 3rd-party cache-coherent devices to a computer system. The resulting new and potentially *different* protocol endpoint implementations of these devices must be able to interoperate with each other and achieve global coherence in flexible system compositions.



(a) A snoop-based, symmetric MESI protocol



(b) Asymmetric specs for 2-node directory-based MESI; the remote side (left) tracks only *M-E-S-I* states, and the home side (right) tracks local and remote states.

Fig. 1: Mealy machines for snoopy and directory-based MESI.

This development motivates both the formal specification of protocol endpoints in such standard, and the means to efficiently and exhaustively test their resulting in-silicon implementations. More concretely, formal specification allows the desired system-level properties to be verified on the abstract compositions of the standard's endpoints. A complete test method then allows those system-level properties to be transferred from the abstract to the composition of successfully tested endpoint implementations. To apply to a standard's complex multi-vendor ecosystem, this test method must operate on the in-silicon implementation only, without requiring access to additional information such as internal documentation or design sources.

Given that the requirements for semi-open-state testing are met, our proposed testing approach is applicable to this setting: a successful test verdict guarantees complete conformance of an in-silicon coherence endpoint to its formal Mealy FSM specification. Semi-open-state testing does not require any additional information beyond access to the in-silicon implementation, and can be naturally integrated into the usual compliance testing workshops conducted for hardware interconnect standards like PCI Express (PCIe).

We will now discuss why the requirements of semi-open-state testing can be met by protocol endpoints of cache coherent interconnect standards. To this end, we take a closer look at the nature of these endpoints. We then argue why the requirements of semi-open-state testing constitute reasonable restrictions on these endpoints and can therefore be imposed on vendor implementations by the standard.

A. Directory-based cache coherence protocols

Basic textbook coherence protocols tend to be *snoopy*: each node can observe the operations performed by all other nodes instantaneously. The classic example is MESI, which associates one of four different states with each line: *M(odified), E(xclusive), S(hared) or I(nvalid)*. The *M* and *E* states imply that the cache holds the only valid copy (dirty or clean resp.) of the data and reads and writes can be performed locally on it without coordination. State *S* implies the copy is valid and clean but may exist in other caches, requiring coordination for writes. A snoopy MESI protocol endpoint can thus be specified with the Mealy FSM in Figure 1a. Transitions are defined on inputs corresponding to local memory requests or snooped-on remote bus transactions, while outputs are initiated bus transactions.

The shared bus required for snoopy protocols does not scale well, and so in practice most real inter-die implementations are *directory-based*, including those used by coherent interconnect standards. These protocols track the cache line status of all participating nodes in a *directory* held at the line's *home node* (typically where the main memory for the line is attached), and coordinate with explicit point-to-point messages instead of bus snooping.

This makes the protocol endpoints asymmetric: remote nodes have the same *stable* states (e.g. *M, E, S, I*) as the snoopy protocol, but the home directory needs stable states that reflect the system-wide state combinations. Moreover, the use of point-to-point messages requires additional *transient states* (sometimes hundreds) in order to cope with all possible interleavings of messages and actions on each node, including conflicting concurrent transactions and message reordering. The resulting two Mealy FSMs are therefore more complex (Figure 1b).

B. Connection to semi-open-state testing

The endpoints of directory-based coherence protocols are amenable to semi-open-state testing: the special visible states in the implementation correspond to the stable protocol states, and the main assumptions we require do hold.

Requirement (1) Testability Hypothesis: A valid *I/O adaptor* can abstract the data in a cache line, assembly load and store instructions, and the format of coherence messages, retaining only the message and software request types. The abstracted behavior of the in-silicon implementation can be *driven* with respect to the inputs of the spec, which requires that the protocol-relevant state does not change except in response to such inputs. Notice that our state machines only reason about a single cache line; silicon implementations generally do indeed treat each line independently [16], although some dependencies may be introduced if the implementation relies on particular message reorderings [26]. For the standard fairness assumption, we further require the implementation's

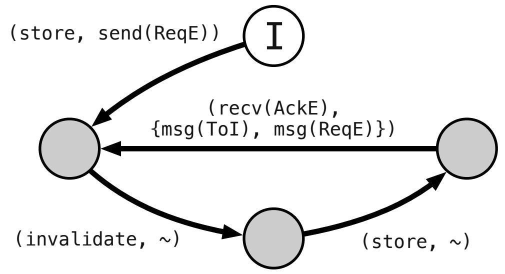


Fig. 2: A cycle of transient states in a remote node, produced by its software repeating concurrent invalidates and reads.

processing of inputs to be *linearizable*. This is usually the intended behavior of a protocol endpoint; whether linearizable processing is achieved needs to be validated separately.

Requirement (2) Visible stable cache line states: Although not used by most programmers, it turns out that existing hardware generally provides precisely this property to software via facilities intended for performance analysis and low-level debugging. An example is the processor we test in section V. Furthermore, the initial state of the protocol naturally corresponds to the *stable Invalid state*, and is therefore among the visible states, as required.

Requirement (3) Stable cache states form a feedback vertex set: To achieve this, our specification must *exclude input buffering* and the remaining *pure protocol processing* must exhibit the feedback vertex property. Without the exclusion of input buffering, a continuous stream of stores and invalidations on the remote node could yield a cycle of transient states, as shown in Figure 2: as soon as coordination with the home node completes, a request is immediately replaced by its buffered successor, and thus no intermediate stable state is visible.

Real implementations do require the buffering behavior we exclude from the specification. However, in practice the buffering only depends on the aspects of the protocol state that the implementation makes directly visible: only entering a stable state serves as a signal to fetch the next request from the buffer. This choice is made by hardware designers precisely to limit complexity of both implementation and validation.

Consequently, we only need pure protocol processing to respect the feedback vertex set property. This requirement might preclude aggressive cache optimizations such as eager replies to requests, or remote-allocate of the line into a remote node's cache (which may require the remote node to handle an unbounded number of home-enforced caching state upgrades and downgrades while a request of its own is pending). In practice, such features are rare.

Requirement (4) Practicability: Recall that the specification needs to be transformed to define its states according to the $\hat{\lambda}$ function (Equation 5). This entails unrolling traces through transient states, which may cause the size of the specification to grow exponentially.

Fortunately, in coherence protocols this is not the case. They exhibit a small upper bound on the number of intermediate transient states, because only a limited number of requests from a peer must be handled before the node can conclude the processing of its own requests and reach the next stable state. In MESI, for example, the home node can only ask the remote to downgrade its state twice (from M/E to S, then from S to I). Furthermore, like other communication protocols, coherence protocols are sparse – only a few messages can be received from a valid peer implementation in any given state. Therefore, we expect the transformation to only yield a moderate increase in size.

V. Application to a real in-silicon implementation

While production hardware for the cache-coherent variants of protocol standards like CXL has yet to appear, we have applied our technique to the cache-coherent interconnect of the Enzian research computer [3]; indeed, this was a motivation for our original work. Enzian can be viewed as a 2-socket NUMA machine combining a Cavium ThunderX-1 48-core ARMv8 CPU with a large Field Programmable Gate Array (FPGA), which also implements the Cavium Coherent Processor Interconnect (CCPI), the CPU's native inter-socket coherence protocol, appearing to the CPU as a second processor node.

While the ThunderX-1 was not originally intended to interoperate at the coherence level with anything other than another ThunderX-1, Enzian was designed to explore the space of emerging coherent heterogeneous platforms, and so must provide an endpoint implementation of CCPI on the FPGA that interoperates with the CPU. In the context of this work, we use the CPU's Last-level cache (LLC) as the system under test, and use a combination of FPGA programming and software running on the CPU to exhaustively test the CPU's silicon implementation of CCPI against our specification.

CCPI is a distributed directory-based MESI coherence protocol whose endpoints satisfy all of the requirements for semi-open-state testing: CCPI maintains cache line independence; it enforces sequential consistency and is able to cope with arbitrary reorderings of messages on the interconnect. Its endpoints further cleanly separate their input buffering and processing, and the design of their processing guarantees that a stable M-E-S-I protocol state (combination) is always reached after at most four transitions. As a result, CCPI's behavior can be accurately described by two Mealy FSMs, one for a remotely-owned and one for a homed cache line, both of which return to a stable state in a bounded number of steps. Furthermore, software on the CPU can use hardware performance counters to determine if a cache line is in a transient or stable state, and for the latter, the state of each cache line can be explicitly read from the cache using privileged registers. Thus, the stable caching states are made visible in the protocol implementation, as required.

Our initial specification of the protocol endpoints was manually constructed from informal vendor documentation, and subsequently refined as a result of the testing process. The result is an exhaustively-tested specification of an in-silicon cache coherence protocol implementation.

A. Testing setup and methodology

Our testing setup operates on a single designated Cache Line Under Test (CLUT), and consists of three components: an *orchestrator* and a C library running on the CPU, and an FPGA testing component. The *orchestrator* is responsible for generating the test stimuli from the supplied specification and coordinating their execution. To generate test stimuli, the orchestrator performs the online graph traversal algorithm by Bourdonov et al. [1] while driving the implementation in tandem. The C library can issue operations (load, store, etc.) on the CLUT and can return the visible aspects of the CLUT's current protocol state (transient or stable with a particular caching state) when invoked by the orchestrator. The FPGA tester implements the underlying reliable link protocol, can send and receive coherence messages to and from the CPU's LLC when directed by the orchestrator, and relay received events back. Communication between the FPGA component and orchestrator cannot use the coherence protocol because it must not interfere with the CLUT state. We exploit the uncached I/O load/store operations that the ThunderX-1 supports as an out-of-band communication channel between the software orchestrator and the FPGA logic.

We must also prevent any other events in the system (such as conflict or capacity misses in the cache) from affecting the CLUT. We achieve this by placing the CLUT in a region of memory otherwise unused, and exploiting a feature of the CPU to "lock" it in the LLC, ensuring that the only operations that affect it are those explicitly initiated by the orchestrator.

This also addresses a further practical problem: since we are deliberately stalling the cache protocol, we run the risk of preventing the orchestrator itself making forward progress unless we can ensure that it does not need to initiate inter-node cache operations. In our current implementation, this can still occasionally happen due to global barrier operations we cannot control, deadlocking the interconnect and causing a "machine check" exception in the processor. In this case, we simply try again: the phenomenon does not affect the validity of a run that completes without a machine check.

B. Experience and results

Having developed our methodology and tools, and based on an incomplete understanding of CCPI derived from vendor documentation, it took approximately 2 person-days to formalize the behavior of the protocol in our specification format. It then took a further person-day to iteratively improve the specification based on testing feedback. A snippet from our specification is shown in Table I. It details some transition the home node can take when the remote node has the CLUT in shared.

s_K		t_U		x	$\rightarrow$	s'_K		t'_U			y
I	S	[	]	SI	$\rightarrow$	I	I	[		]	~
I	S	[	]	SE$_d$	$\rightarrow$	I	E	[		]	AE
I	S	[	]	LE	$\rightarrow$	I	S	[	(LE/IV)	]	IV
I	S	[	]	IE$_d$	$\rightarrow$	I	S	[	(IE$_d$/~)	]	~

TABLE I: Excerpt from our home node specification: to the left of the arrow we denote the (state, input) pair consisting of last stable state s_K and the trace t_U since. To the right of the arrow we denote the state (s'_K, t'_U) the machine transitions to, and the output y generated. The symbol ~ denotes that no output is generated.

The resulting remote node specification yields a successful verdict, establishing that the ThunderX-1's CCPI remote node implementation is generalized-quasi-equivalent to this specification. This final specification has 107 transitions between the 4 stable M-E-S-I states and 42 additional transient states.

In an interoperability scenario, creating the specification would be done at most once against a "gold standard" reference implementation, or would ideally be provided directly by the coherent interconnect standard.

Subsequent *conformance testing* against the specification is much quicker. Testing one transition of the ThunderX-1's in-silicon implementation takes approximately 10 milliseconds, most of which is spent waiting between applying the input and observing the generated outputs to ensure that the ThunderX-1 has finished the processing of the former. Exhaustive testing of the remote node specification concludes in under two minutes, during which each stimulus is executed multiple times to exercise all potential non-deterministic behavior.

Moreover, our specification is human-readable and comprehensible, in part because it obviates the need for abstract state identifiers. With abstract identifiers, a reader would have to explicitly remember the context of each and every such identifier, something challenging for a protocol of this complexity. In contrast, our trace-based identifiers carry all the relevant context to determine at a glance which sequence of events has lead to that state, and eliminate the need to think about how to group behaviors into states, since every behavior has a unique representation in the specification.

We are convinced that the testing accurately reflects the implementation behavior. All the discrepancies revealed when iterating our manually-written specification were reasonable from a protocol perspective. We also uncovered behavior that we could not have known based on the documentation; for example, an undocumented error message that the remote node generates when it receives unexpected messages in some situations, or the elision of a particular protocol message in a way that correctly maintains coherence, yet is at odds with convention in the rest of the protocol implementation.

Finally, we observe completely deterministic behavior in testing. If our modelling had missed an important aspect of the protocol, we would expect this to manifest as non-deterministic behavior.

VI. Conclusion and future work

Our experience shows that semi-open-state testing is a viable approach for testing state machine implementations. Building on existing open-state testing techniques allows us to design an efficient testing procedure, covering a significant space of real-world state machines. We evaluate these claims by testing the in-silicon state machine of a ThunderX-1 LLC against a specification of a directory-based cache coherence protocol. Exhaustively testing the ThunderX-1 coherence implementation completes in a matter of minutes, demonstrating that our approach is also efficient in practice.

Our method can extend to specification *synthesis*, where we extract the behavior of a state machine implementation from an implementation. This is useful in situations where we observe interactions between implementations known to be good (by accident or design). Synthesis helps derive specifications where we only have access to implementations, e.g. when observing the interactions of two reference implementations.

The ThunderX-1 specification we test is restricted to two protocol actors. Proposed coherent interconnect standards like CXL allow many actors participating in a coherence protocol, and will require more sophisticated specifications to deal with the additional complexity. Testing whether the composition of heterogeneous cache coherence implementations provides cache coherence correctly is an important consideration for system integrators and hardware designers alike.

There are other real-world protocols that could benefit from semi-open-state testing, for example remote conformance of TCP stacks. Successful application of our approach in the context of other protocols and specifications would further demonstrate its general applicability.

Semi-open-state testing extends the set of state machine implementations that can be efficiently but exhaustively tested, to the case where a stable subset of implementation states is observable. We successfully apply semi-open-state testing to the in-silicon implementation of unmodified, real hardware.

VII. Acknowledgements

We thank the anonymous reviewers for their helpful comments and feedback, and we are grateful to Google for financial support.

References

[1] I. B. Bourdonov and A. S. Kossatchev, "Complete open-state testing of limitedly nondeterministic systems," *Programming and Computer Software*, vol. 35, no. 6, pp. 301–313, Nov. 2009. [Online]. Available: https://doi.org/10.1134/S0361768809060012

[2] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, May 1978. [Online]. Available: https://doi.org/10.1109/TSE.1978.231496

[3] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An open, general, CPU/FPGA platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Feb. 2022, pp. 434–451. [Online]. Available: https://doi.org/10.1145/3503222.3507742

[4] D. D. Sharma and I. Agarwal, "Compute Express Link 3.0 Standard," CXL Consortium, Tech. Rep., 2022.

[5] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996. [Online]. Available: https://doi.org/10.1109/5.533956

[6] R. M. Hierons, "Testing from Partial Finite State Machines without Harmonised Traces," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1033–1043, Nov. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2652457

[7] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 109–124. [Online]. Available: https://doi.org/10.1145/186258.187153

[8] M.-C. Gaudel, "Software testing based on formal specification," in *Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*. Springer Berlin Heidelberg, 2010, pp. 215–242. [Online]. Available: https://doi.org/10.1007/978-3-642-14335-9_7

[9] R. M. Hierons, "FSM quasi-equivalence testing via reduction and observing absences," *Science of Computer Programming*, vol. 177, pp. 1–18, May 2019. [Online]. Available: https://doi.org/10.1016/j.scico.2019.03.004

[10] F. Aarts, B. Jonsson, and J. Uijen, "Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction," in *Testing Software and Systems*, A. Petrenko, A. Simão, and J. C. Maldonado, Eds. Berlin, Heidelberg: Springer, 2010, pp. 188–204. [Online]. Available: https://doi.org/10.1007/978-3-642-16573-3_14

[11] E. F. Moore *et al.*, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.

[12] A. Dahbura, K. Sabnani, and M. Uyar, "Formal methods for generating protocol conformance test sequences," *Proceedings of the IEEE*, vol. 78, no. 8, pp. 1317–1326, Aug. 1990. [Online]. Available: https://doi.org/10.1109/5.58319

[13] D. Sidhu and T.-K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 413–426, Apr. 1989. [Online]. Available: https://doi.org/10.1109/32.16602

[14] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, "Nondeterministic State Machines in Protocol Conformance Testing." in *Proceedings of the IFIP TC6/WG6. 1 Sixth International Workshop on Protocol Test systems VI*, Jan. 1993, pp. 363–378.

[15] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 1998.

[16] F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Computing Surveys*, vol. 29, no. 1, pp. 82–126, Mar. 1997. [Online]. Available: https://doi.org/10.1145/248621.248624

[17] ——, "Formal verification of complex coherence protocols using symbolic state models," *Journal of the ACM*, vol. 45, no. 4, pp. 557–587, Jul. 1998. [Online]. Available: https://doi.org/10.1145/285055.285057

[18] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 247–260. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00030

[19] H. Kahlouche, C. Viho, and M. Zendri, "Hardware Testing Using a Communication Protocol Conformance Testing Tool," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer, 1999, pp. 315–329. [Online]. Available: https://doi.org/10.1007/3-540-49059-0_22

[20] A. Kriouile and W. Serwe, "Using a formal model to improve verification of a cache-coherent system-on-chip," in *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, 2015, pp. 708–722.

[21] J. You, D. Bural, J. Brown, and J. Zbiciak, "Red Baron: Near/post-silicon SoC cache coherence stress tester," in *2016 IEEE Dallas Circuits and Systems Conference (DCAS)*, Oct. 2016, pp. 1–4. [Online]. Available: https://doi.org/10.1109/DCAS.2016.7791135

[22] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *2008 IEEE International Conference on Computer Design*, Oct. 2008, pp. 348–355. [Online]. Available: https://doi.org/10.1109/ICCD.2008.4751884

[23] D. Foley and J. Danskin, "Ultra-Performance Pascal GPU and NVLink Interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[24] CCIX Consortium and others, "Cache Coherent Interconnect for Accelerators (CCIX)," January 2019. [Online]. Available: http://www.ccixconsortium.com

[25] W. W. Terpstra, “TileLink: A free and open-source, high-performance scalable cache-coherent fabric designed for RISC-V,” in *Proc. 7th RISC-V Workshop*, 2017.

[26] M. Martin, “Formal verification and its impact on the snooping versus directory protocol debate,” in *2005 International Conference on Computer Design*, Oct. 2005, pp. 543–549. [Online]. Available: https://doi.org/10.1109/ICCD.2005.58

Formal Methods in Computer-Aided Design 2024

Memory Consistency Model-Aware Cache Coherence for Heterogeneous Hardware

Rachel Cleaveland
Stanford University
Stanford, CA, USA
rcleavel@stanford.edu

Caroline Trippel
Stanford University
Stanford, CA, USA
trippel@stanford.edu

***Abstract*—Implementing cache-coherent shared memory in heterogeneous systems is challenged by memory consistency model (MCM) mismatches among compute elements: *what* the system-wide MCM should be and *how* it should be enforced are not well-defined. In this paper, we posit that C11—the seminal heterogeneous MCM—is the natural MCM choice for such systems. Based on this philosophy, we design and verify MEMGLUE, an update-based *consistency protocol* (i.e., an MCM-respecting coherence protocol) that enforces a slight strengthening of C11 among a set of interacting heterogeneous compute clusters.**

MEMGLUE has three notable features. First, it is *modular*: any cluster equipped with a MEMGLUE translation *shim* can "plug into" any MEMGLUE system. Second, it is *verifiable*: one system-wide proof ensures that MEMGLUE upholds the C11 MCM with respect to MEMGLUE messages exchanged by clusters' shims, and per-cluster proofs ensure that shims correctly translate relevant cluster coherence protocol messages to MEMGLUE protocol messages. Third, it is *polite*: MEMGLUE is compatible with a wide range of cluster coherence protocols and MCMs and exploits the permissible relaxed ordering behavior of each cluster to a high degree.

I. INTRODUCTION

Modern computer systems are increasingly heterogeneous: outsourcing computation from general-purpose CPUs to special-purpose hardware increases computational throughput while saving power [36], [25], [37]. And, as evidenced by the emergence of several industrial designs and standards (e.g., NVLink-C2C [5], CXL [71], CAPI [73], CHI [12], HSA [31], CCIX [2]), there is growing interest in implementing *cache-coherent shared memory* in such systems. Allowing components to share a coherent address space eases the burden of explicit memory management while reducing intra-system data movement and increasing performance [71], [42], [48], [58].

Unfortunately, implementing cache-coherent shared memory in heterogeneous systems is not straightforward. A core issue is that the heterogeneous processing elements comprising modern Systems-on-Chip (SoCs) [37], multi-chiplet designs [84], [3], and data centers [26], [22], [76], [40] feature disparate *memory consistency models* (MCMs) across their *instruction set architectures* (ISAs). That is, these processing elements assume/enforce differing restrictions on the ordering and visibility of (*all*) shared memory accesses [59]. Traditional coherence invariants order *same-address* memory accesses only [59]. Failure to also coordinate ordering among *different-address* accesses in heterogeneous shared memory systems can lead to unexpected program outcomes [52], [63], [34], [9].

Most proposals for implementing heterogeneous cache-coherent shared memory today require hardware designers and software developers to collaboratively manage MCM diversity per system [14], [66], [5], [71].

Recent academic work shows that software developer burden can be alleviated by offloading the task of managing MCM heterogeneity to hardware *consistency protocols*. And, hardware designer effort can be reduced by automatically synthesizing a hardware consistency protocol per set of heterogeneous *clusters*, given per-cluster coherence protocol and MCM specifications as input [63]. (In this paper, a cluster denotes a group of *homogeneous* compute elements sharing a memory hierarchy.) Yet, the following key challenges remain.

First, synthesized consistency protocol implementations and the system-wide MCMs they intend to enforce (described as the "amalgamation" of the per-cluster MCMs [63]) are *unique* for distinct inputs to the synthesis procedure. Each protocol-MCM pair requires verification to ensure that synthesis did not unintentionally introduce protocol bugs—a notoriously difficult task [21], [65], [82], [15], [49], [55], [51], [54], [75], [53]. Second, deploying these consistency protocols requires explicitly merging, and thus modifying, cluster coherence protocol implementations (adding new transient states) and cache structures (combining clusters' directory controllers and last-level caches). This strategy is not readily compatible with systems where some cluster's memory system cannot be co-designed with the rest (e.g., SoCs/multi-chiplet designs with third-party cores/chiplets, data-centers). Third, clusters with non-multiple-copy-atomic (non-MCA) MCMs [61], [50], [10], [38] and update-based coherence protocols are not supported.

A. This Paper

Towards resolving the challenges above, we propose MEMGLUE, a universal hardware consistency protocol that is *verifiable*, *modular*, and *polite*. We coin the term *consistency protocol* to refer to an MCM-respecting coherence protocol, which provides coherent shared memory for arbitrary sets of heterogeneous clusters while upholding the memory ordering requirements of their respective ISA MCMs.

Insight 1: A universal consistency protocol enables verifiability *and* modularity*:* MEMGLUE is a *universal* consistency

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_22

protocol, meaning that its implementation and the system-wide MCM it enforces are the same for any composition of heterogeneous clusters. In particular, MEMGLUE enforces a slight strengthening of the C11/C++11 MCM [39], [46]—henceforth referred to as C11—among clusters. This is a natural design choice, since C11 was explicitly intended to be compatible with the breadth of modern ISA MCMs, and many new ISAs consider C11-compatibility a core requirement [78], [50], [80]. C11 defines a variety of memory and synchronization operations with different ordering *strengths* [4] attached to them, so as to closely match the ordering semantics of a wide range of ISA memory and synchronization instructions. MEMGLUE protocol messages directly adopt the syntax and semantics of these variable-strength C11 operations.

Per-cluster MEMGLUE translation *shims* translate relevant cluster coherence protocol messages to MEMGLUE protocol messages.[1] For correctness, a shim must consider the ordering requirements of the ISA instruction(s) that generate a particular coherence protocol message and output a MEMGLUE message of equal (or greater) ordering strength. However, shim design and its verification can be simplified thanks to MEMGLUE's C11-centric design and the existence of formally verified compiler mappings from C11 to a variety of ISA MCMs [69]. In particular, a shim can select the strongest C11 operation that could have generated a given coherence protocol message, and output its matching MEMGLUE message.

Any cluster equipped with a MEMGLUE shim can "plug into" any MEMGLUE system and know that its memory ordering behaviors will adhere to its ISA MCM. Moreover, the MEMGLUE protocol, which coordinates C11-style operations among clusters' shims, need only be designed and verified *once*. Shims are verified once per cluster, but are simpler and can reuse C11 compiler mappings [69]. By decoupling per-cluster translation logic from its system-wide protocol implementation, MEMGLUE adopts a modular design philosophy, which avoids explicitly merging clusters' memory systems.

Insight 2: Update-based protocols enable politeness*:* Ideally, a consistency protocol should be compatible with as many local cluster coherence protocols and MCMs as possible, while retaining performance of intra-cluster shared memory communication (compared to a homogeneous shared memory system), and maximizing performance of inter-cluster communication. We say that such a protocol is *polite*. Our goal of politeness, combined with the producer-consumer access patterns typically seen among clusters in heterogeneous systems [72], [79], motivates us to implement MEMGLUE as an *update-based* protocol [35] (related work adopts an *invalidation-based* approach [62], [9], [71], [63], [34]).

We show that as the fraction of clusters with weak (local) MCMs grows in a MEMGLUE system, so does the number of observable heterogeneous program execution behaviors (§VI-A). This means that MEMGLUE effectively exploits the permissible relaxed ordering behavior of its clusters' MCMs.

[1]A "shim" is "a thin piece of wood, rubber, metal, etc. which is thicker at one end than the other, that you use to fill a space between two things that do not fit well together" [67].

Thread 1	Thread 2
1: Wx = 1 2: Wy = 1	3: Ry = 1 4: Rx = 0

Fig. 1: Message Passing (MP) litmus test. Memory locations contain zero initially. The outcome is permitted by some MCMs (e.g., ARMv8 [11]), but not others (e.g., x86-TSO [8]).

Plus, MEMGLUE accommodates cluster coherence protocols and MCMs that current approaches for designing heterogeneous MCMs [63], [34] and their implementations as consistency protocols [63] do not, e.g., update-based protocols [63], [34], and non-multiple-copy-atomic (non-MCA) MCMs [63].

We summarize our contributions as follows.

- **MEMGLUE Approach.** We propose MEMGLUE, a universal consistency protocol for heterogeneous shared-memory systems. To our knowledge, MEMGLUE represents the only attempt to go beyond an operational model and implement C11 directly as a hardware protocol.
- **MEMGLUE Design.** We design MEMGLUE as an update-based protocol to accommodate clusters with a variety of local coherence protocols and MCMs and to optimize for inter-cluster producer-consumer communication patterns.
- **MEMGLUE Murφ Model.** We implement MEMGLUE in the Murφ model checker and prove that it *closely* upholds C11 for a suite of 6,738 litmus test programs.
- **MEMGLUE Correctness Proof.** We manually prove that MEMGLUE upholds C11 for all programs.

II. BACKGROUND AND MOTIVATION

Memory consistency models (MCMs) govern the ordering and visibility of shared memory accesses in parallel programs [59]. They define what program *executions*, and thus *outcomes* (mappings of a program's shared memory loads to the values they return), are permitted/forbidden. For the same program, one MCM may permit an outcome that another forbids. Such distinctions are often captured using small parallel programs, called *litmus tests* (Fig. 1).

MCMs span the hardware-software stack: from high-level languages (HLLs) [19], [56], [18] to intermediate representations (IRs) [83] and ISAs [50], [64], [78], [11], [38]. Yet, for HLL programs, the HLL MCM ultimately dictates which of its executions are permitted, and compilation to IRs and/or ISAs must avoid creating more permitted execution possibilities.

A. Memory Consistency Model Overview

MCMs are often specified axiomatically [8], [18], [46], [50], [68] by defining a "happens-before" relation ($\rightarrow_{hb}$) between instructions that restricts which executions are permitted. Permitted executions are those *not* containing happens-before cycles. In Fig. 1, for example, a *strong* MCM may instantiate: $1 \rightarrow_{hb} 2 \rightarrow_{hb} 3 \rightarrow_{hb} 4 \rightarrow_{hb} 1$ (i.e., 1 happens-before 2, etc.). This cycle implies a contradiction—instruction 1 happened-before itself—indicating that the MCM disallows this execution. A *weaker* MCM may only instantiate $2 \rightarrow_{hb} 3$ and $4 \rightarrow_{hb} 1$, resulting in an acyclic execution that is permitted.

$$
\begin{aligned}
\texttt{rs} &= [\texttt{W}]\ ;\ (\texttt{sb\&loc})?\ ;\ [\texttt{W\&} \sim \texttt{NA}]\ ;\ (\texttt{rf};\texttt{rmw})* \\
\texttt{sw} &= [\texttt{REL} \mid \texttt{ACQREL} \mid \texttt{SC}]\ ;\ ([\texttt{F}];\texttt{sb})?\ ;\ \texttt{rs}\ ;\ \texttt{rf}\ ; \\
&\qquad [\texttt{R\&} \sim \texttt{NA}]\ ;\ (\texttt{sb};[\texttt{F}])?\ ;\ [\texttt{ACQ} \mid \texttt{ACQREL} \mid \texttt{SC}] \\
\texttt{hb} &= (\texttt{sb} \mid \texttt{sw})+ \\
\texttt{eco} &= \texttt{rf} \mid \texttt{mo} \mid \texttt{fr} \mid \texttt{mo};\texttt{rf} \mid \texttt{fr};\texttt{rf} \\
\texttt{scb} &= \texttt{hb} \mid \texttt{mo} \mid \texttt{fr} \\
\texttt{psc\_base} &= ([\texttt{SC}] \mid [\texttt{F\&SC}]\ ;\ \texttt{hb}?)\ ;\ \texttt{scb}\ ;\ ([\texttt{SC}] \mid \texttt{hb}?\ ;\ [\texttt{F\&SC}]) \\
\texttt{psc\_f} &= [\texttt{F\&SC}]\ ;\ (\texttt{hb} \mid \texttt{hb};\texttt{eco};\texttt{hb})\ ;\ [\texttt{F\&SC}] \\
\texttt{psc} &= \texttt{psc\_base} \mid \texttt{psc\_f}
\end{aligned}
$$

Fig. 2: A subset of the C11 derived relations [46]. The $|$, $;$, $?$, $+$, and $*$ relational operators represent union, sequencing, union with the identity relation, transitive closure, and reflexive transitive closure. W, R, and F represent write, read, and fence instructions. ACQ and REL denote instructions with the C11 acquire and release memory orders, and so on.

B. The C11 Memory Consistency Model

MemGlue targets C11—the seminal heterogeneous MCM [19]—and more specifically, the RC11 variant [46]. From this point on, we use "C11" to refer to RC11, unless otherwise stated. C11 programs are intended to be compiled to and executed on virtually any hardware, despite the fact that each target ISA has its own MCM.

To leverage weak ISA MCM offerings, C11 provides several *memory orders* [4] (or "strengths") for each memory and fence operation: *relaxed* (RLX), *acquire* (ACQ), *release* (REL), *acquire-release* (ACQREL), and *sequentially consistent* (SC).[2] Programmers may label writes as RLX, REL, or SC; reads as RLX, ACQ, or SC; and fences as ACQ, REL, ACQREL, or SC. The read and write components of atomic "read-modify-write" (RMW) operations can take on read and write labels, respectively, yielding RLX, ACQ, REL, ACQREL, or SC RMWs. The strength of each memory order subsumes that of all weaker orders, per the partial order RLX $\prec$ REL/ACQ $\prec$ ACQREL $\prec$ SC. So, any ordering restrictions on RLX instructions will apply to REL instructions, and so on. Note that this partial order does not relate REL to ACQ, but both are stronger than RLX and weaker than ACQREL. Compilers translate these "labeled" C11 operations into sequences of load, store, and fence instructions in the target ISA, such that the C11 ordering guarantees will be upheld when compiled programs run on hardware [69].

As illustrated in Figs. 4, 5 and 6, C11 defines a variety of relations between operations, which it uses to specify legal program outcomes. *Base relations* include:

- sb (sequenced-before): describes program order.
- rf (reads-from): relates writes to same-address reads that read from them.
- mo (modification order): relates same-address writes in the order that they commit to memory.
- fr (from-reads): relates a read to a "newer" same-address write that happened mo-after the write that it read from.

[2]RC11 does not support C11's *consume* memory order, as it is not used by major compilers [46].

$$
\begin{aligned}
\texttt{Coherence} &= \texttt{irreflexive}\,(\texttt{hb}\ ;\ \texttt{eco}) \\
\texttt{SC} &= \texttt{acyclic}\,(\texttt{psc}) \\
\texttt{Atomicity} &= \texttt{rmw} \cap (\texttt{fr};\texttt{mo}) = \emptyset \\
\texttt{No-Thin-Air} &= \texttt{acyclic}\,(\texttt{sb} \mid \texttt{rf})
\end{aligned}
$$

Fig. 3: C11 axioms [46].

- rmw (read-modify-write): relates the read component of an RMW to the write component.

Note that sb, rf, mo, and fr are execution-specific, with sb encoding a program's dynamic control-flow and the others encoding its data-flow. The remaining C11 relations (Fig. 2) are derived from these base relations.

Two notable *derived relations* are sw (synchronizes-with) and hb (happens-before): hb is the union of sw and sb, and sw relates *release operations* to *acquire operations*.[3] For example, as shown in Fig. 4a, when a release write (or any write that is sb-after a same-address release write) is related to an acquire read by rf, the release write is *also* related to the acquire read by sw. The sw relation can also involve fences. A release fence that is sb-before a write may feature an outgoing sw edge, and an acquire fence that is sb-after a read may feature an incoming sw edge. As shown in Fig. 4b, an rf edge between such a write and such a read instantiates an sw edge between their corresponding release and acquire fences. Note that release fences (release writes) can also be related to acquire reads (acquire fences) by sw.

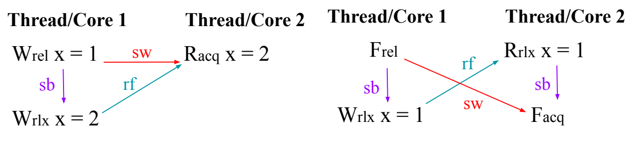
The different C11 memory orders induce different relations between program operations, and thus different constraints on permitted program executions. RLX operations are subject to few restrictions, only guaranteeing *atomicity* (i.e., partially-performed writes cannot be observed) and *coherence* (i.e., all threads can agree on a total order in which same-address memory operations take place, or *SC-per-location* [47]). REL and ACQ operations further restrict legal program outcomes by requiring that all operations visible to a release must be visible to any acquire that is related to the release by sw. This requirement renders the outcome in Fig. 5a forbidden. Finally, all threads must agree on a total order in which SC operations take place. That is, SC reads on different threads may not disagree on the order of SC writes.

Overall, C11 defines legal program executions using four axioms (Fig. 3): Coherence, SC, Atomicity, No-Thin-Air.

The Coherence axiom states that pairs of operations related by hb in one direction may not be related by eco (Fig. 2) in the opposite direction. It enforces (among other things) SC-per-location. Fig. 5 shows how the outcome in Fig. 1 can be forbidden (5a) or permitted (5b) by Coherence depending on the strengths of the program's C11 operations and how they instantiate hb.

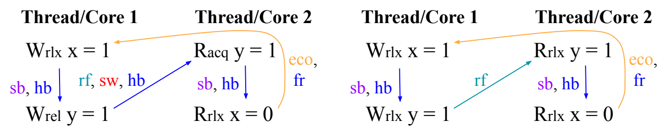
The SC axiom asserts that SC operations must be totally ordered. Fig. 6 shows the Independent Readers, Independent

[3]"Release operations" ("acquire operations") denote memory and fence operations whose strengths are at least as strong as REL (ACQ) per §II-B.



(a) Without fences. (b) With fences.

Fig. 4: Example instantiations of the `sw` relation.



(a) `REL-ACQ` (Forbidden) (b) No `REL-ACQ` (Permitted)

Fig. 5: MP litmus test variants. The `REL-ACQ` synchronization induces `hb` in Fig. 5a, resulting in a violation of `Coherence`.

Writers (IRIW) litmus test, which highlights this requirement and distinguishes orderings enforced by `REL-ACQ` synchronization (when `sw` involves at least one non-`SC` operation) versus `SC-SC` synchronization (when `sw` involves two `SC` operations).

`Atomicity` forbids intervening writes between any read/write pair related by `rmw`.

Compared to the original C11 [19], RC11 fixes issues with `SC` semantics and adds the `No-Thin-Air` axiom. `No-Thin-Air` requires that an execution cannot speculatively evaluate a read operation, such that this speculation satisfies itself through a cyclic chain of dependence [20]. That is, values cannot appear "out-of-thin-air."

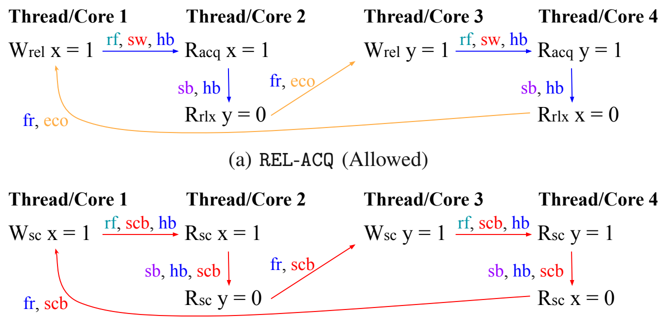
In practice, MEMGLUE implements a slight strengthening of RC11 that uses a *strictly stronger* version of the `scb` relation, which appears in the `SC` axiom [18]. RC11 weakened the `scb` relation to accommodate Power processors, which can produce outcomes that violate the `SC` axiom under the stronger `scb` definition when programs mix `SC` and non-`SC` operations [46]. For MEMGLUE's purposes, implementing a strictly stronger variant of RC11 implies that RC11 is upheld. Further, doing so reduces MEMGLUE's metadata requirement without deviating too far from (the weaker) RC11 behavior (§VI-A).

C. Update-Based Cache Coherence Protocols

Coherence protocols may be *invalidation-* or *update*-based, and our MEMGLUE implementation adopts the latter approach.

Invalidation-based protocols require caches to send invalidation requests to remote cores when they want to perform a write. When a core wants to read a cache line that has been invalidated, it must request access to it through the protocol. Invalidation-based protocols often maintain the *single-writer, multiple-reader* (SWMR) invariant by requiring that all remote copies of a line be invalidated *before* a write may perform [59].

Update-based protocols [13], [74] replace invalidations with updates that propagate writes to remote cores as soon as they perform locally, trading off lower read latency for higher network traffic. In general, each cache write results in a message sent to all sharers, but remote cores always have the most up-to-date data and can thus perform reads immediately [35].



(a) `REL-ACQ` (Allowed)

(b) `SC-SC` (Forbidden)

Fig. 6: IRIW litmus test that is forbidden by C11 iff all the operations are `SC`, due to a cycle in `scb` that violates `SC`.

III. MEMGLUE PRELIMINARIES

We now give an overview of the MEMGLUE consistency protocol, before presenting two implementations (in §IV and §V) that make different assumptions about interconnection network ordering guarantees.

A. MEMGLUE Overview

MEMGLUE is an update-based consistency protocol that coordinates correct shared memory interactions among a heterogeneous set of compute clusters, which have been augmented with MEMGLUE translation shims (Fig. 7).

The MEMGLUE protocol operates within a fragment of C11 that includes `RLX`, `ACQ`, `REL`, and `SC` memory operations (including the read and write components of RMWs) and `SC` fences. We omit support for the strictly weaker `REL`, `ACQ`, and `ACQREL` fences for now. The semantics of RMWs is described in our manual proof in our open-source repository [1]. However, RMWs are not implemented in our Murφ models, nor are they discussed in the paper for space reasons.

MEMGLUE shims intercept relevant coherence protocol messages internal to their local clusters and, based on their local ISA MCMs, generate C11-style messages (§III-B) to be handled by the MEMGLUE protocol.

We justify our decision to implement MEMGLUE as a novel (§III-A2) update-based (§III-A1) protocol below.

1) Why Update-based Consistency Protocols?: Recall our goal of designing a heterogeneous consistency protocol that is *polite* (§I). That is, MEMGLUE should not overly-restrict clusters' (i) coherence protocols (invalidation- or update-based variants should be supported), (ii) MCMs (any MCM should be supported), (iii) performance on intra-cluster shared memory communication (operations on memory locations shared within a single cluster should perform comparably to when the cluster is not plugged into a MEMGLUE system), and (iv) performance on inter-cluster shared memory communication.

Requirement (iv) precludes consistency protocols that enforce SWMR (§II-C) among clusters, which subject intercluster communication to sequentially consistent ordering constraints [47], [59], [57]. Update-based protocols generally do not uphold SWMR [35], nor do certain invalidation-based

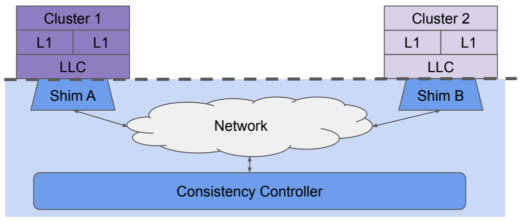


Fig. 7: MEMGLUE system with two heterogeneous clusters. MEMGLUE operates below the dashed line.

protocols, such as those that permit delayed invalidations [43], [44]. Such protocols are reasonable options for exploiting permissible relaxed ordering behaviors between clusters. However, for the reasons below, we elect to implement MEMGLUE as an update-based protocol.

Related to requirements (i) and (ii), we find that update-based consistency protocols easily support both update- and invalidation-based cluster coherence protocols (§VI-A demonstrates the latter), and enable a C11-centric design that can accommodate arbitrary cluster MCMs.

Related to requirement (iv), MEMGLUE orchestrates inter-cluster communication, which we anticipate to largely feature producer-consumer access patterns (e.g., a producer/consumer cluster writes to/reads from a shared queue [79]). For this style of communication, update-based protocols have been shown to perform better than invalidation-based alternatives [23]. One may initially worry about increased memory traffic between clusters. However, MEMGLUE is compatible with several performance optimizations for update-based protocols that reduce network traffic (e.g., an exclusive state [13], [74], competitive updates [35]). Plus, thread migration, which exacerbates update traffic in homogeneous shared memory systems [33], [35], is unlikely across heterogeneous MEMGLUE clusters.

2) Why a Novel Protocol?: We implement MEMGLUE as a novel protocol for two main reasons. First, our MEMGLUE implementation can be viewed as an abstract-machine operational model of (a slight strengthening of) C11. Due to its complexity, such a model for C11 has not yet been developed [30], [41], [60], and no existing coherence protocol comes close to approximating C11 behavior. Second, many prior update-based coherence protocols make restrictive assumptions about the orderedness of the network through which messages travel [74], [13], [81], [33], which we wish to avoid.

B. MEMGLUE *Hardware Primitives*

MEMGLUE introduces two types of hardware structures to mediate communication between clusters: per-cluster *shims* and a single system-wide *consistency controller* (CC).

Shims interface between local clusters and the MEMGLUE system. Within their local clusters, shims intercept relevant coherence protocol messages that are exchanged on behalf of ISA write, read, and fence instructions; translate them into their C11 analogs (§IV-D); and send `WRITE`, `RREQ`, and `FREQ` MEMGLUE messages, respectively, to the CC. From the CC, shims can receive `WRITE`, `WRITE_ACK`, `RRESP`, `FREQ`, and `FRESP` messages (discussed in §IV-A). MEMGLUE messages contain different metadata to ensure they are correctly ordered by the protocol, such as C11-style strengths (`RLX`, `REL`, `ACQ`, or `SC`). The CC acts as the directory structure within MEMGLUE: all messages from the shims are sent to the CC, which orders, responds to, and reroutes them appropriately.

	Remote Propagation	Local Propagation
MSI + TSO $\texttt{ppo} = \texttt{sb} \setminus (\texttt{W},\texttt{R})$	L1 cache write hit	Invalidate all locally cached copies. Write data to LLC.
Firefly + SC $\texttt{ppo} = \texttt{sb}$	Shared bus write	Place write update onto the shared bus.
RCC + RC $\texttt{ppo} = \texttt{sb} \setminus ((\texttt{RLX},\texttt{RLX}) \cup (\texttt{REL},\texttt{RLX}) \cup (\texttt{RLX},\texttt{ACQ}))$	Shared L2 cache write back	Write data to shared L2 cache.

TABLE I: Local and remote write propagation strategies for shim integration for several protocols: MSI (invalidation-based) [59], Firefly (update-based) [74], and RCC (self-invalidation-based) [59]. Remote Propagation provides the local coherence actions that trigger a shim to send a `WRITE` update to the CC; Local Propagation provides the actions that a shim performs to propagate `WRITE` updates within its cluster.

The shims and CC track additional metadata per valid cache line in existing cluster caches. A shim maintains a metadata cache that shadows its cluster's shared last-level cache (LLC). Without loss of generality, we assume inclusive cluster LLCs. The CC acts as a directory for the full heterogeneous MEMGLUE system, maintaining data, metadata, and cluster-granularity sharer lists per cache line present in any of its clusters' LLCs. A cache line tracked by a shim is invalid (valid) if its corresponding LLC cache line is invalid (valid). A cache line in the CC is invalid (valid) if it is invalid (valid) in every (some) cluster's LLC. The shims and CC also track a timestamp per cache line (§IV-B1).

C. *Write Propagation and Shim Integration*

Equipping a cluster with a MEMGLUE shim requires determining (i) when intra-cluster operations should be communicated to remote clusters, and (ii) how MEMGLUE operations arriving from a remote cluster should be propagated internally.

The answer to (i), in short, depends primarily on what intra-thread write-write orderings the local MCM globally enforces.

The cluster actions which require external communication are cache updates (shims must update remote clusters), cache misses (shims must retrieve data and/or metadata), and fences (shims must synchronize with remote clusters). When the shims observe local coherence protocol messages indicative of these actions, they send `WRITE`, `RREQ`, and `FREQ` messages to the CC, respectively. Usually, `WRITE` and `FREQ` messages correspond to (committed) ISA instructions within the local cluster, so if a cluster's MCM globally orders a pair of such instructions, its shim must send their generated MEMGLUE messages to the CC in the same order. For most cluster coherence protocols, where ISA fences do not generate protocol messages, the main task of a MEMGLUE shim is to preserve *globally-enforced* orderings among its cluster's ISA writes.

Such globally-enforced orderings may order writes in different threads (inter-thread) or the same thread (intra-thread). For clusters with MCA MCMs, intra-thread orderings are typically captured by a *preserved program order* (ppo) relation [8]. For clusters with nMCA MCMs, they are often embedded in more subtle causality relations [50]. A shim can observe *inter-thread* write-write orderings (e.g., mo) at a cluster's coherence ordering point; however, globally-enforced *intra-thread* write-write orderings may require shims to be placed higher (closer to cluster cores) within the memory hierarchy.

To see how a cluster's intra-thread write-write ordering requirements inform shim placement, consider the following *total store order* (TSO) [70] and *release consistency* (RC) [59], [32] examples from the *Remote Propagation* column of Table I. For a TSO cluster, intra-thread $\mathtt{W} \rightarrow_{\mathtt{sb}} \mathtt{W}$ ordering is preserved globally. Hence, its shim must send out a WRITE message upon each L1 write hit and therefore monitor all L1 cache interfaces. In contrast, an RC cluster preserves intra-thread $\mathtt{W} \rightarrow_{\mathtt{sb}} \mathtt{W}_{\mathtt{rel}}$ order globally, but not intra-thread order among non-rel writes.[4] When a $\mathtt{W}_{\mathtt{rel}}$ is performed at a core, all dirty data in the L1 are written back to the shared L2 (the cluster's LLC) before the $\mathtt{W}_{\mathtt{rel}}$ itself is written back to the L2. Thus, the shim need only monitor the cluster's L2 interface.

To question (ii), MEMGLUE propagates incoming WRITE messages (which carry updates from remote clusters) within a cluster by leveraging its local coherence protocol. The *Local Propagation* column of Table I gives examples.

IV. Ordered MemGlue Consistency Protocol

We first present Ordered MEMGLUE (MEMGLUE$_{\text{O}}$), which assumes an ordered interconnection network (i.e., messages from the same sender to the same receiver arrive in the order they were sent). A complete specification of the protocol can be found in our open-source repository [1].

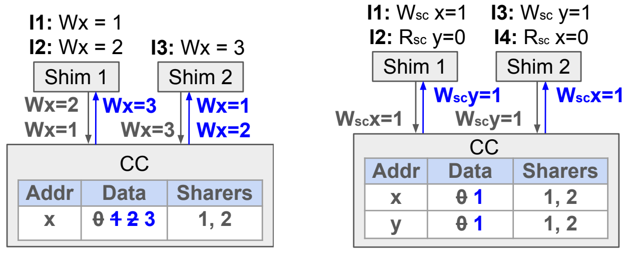
A. MemGlue$_{\text{O}}$ *Protocol*

In this section, we present a simplistic view of MEMGLUE's actions upon observing cluster-local instructions via their induced coherence protocol messages. In §IV-B we refine the MEMGLUE protocol to maintain the C11 axioms (§II-B).

Cluster writes. When a shim sees a cluster write (via a write hit or write-back, §III-C), it immediately sends a WRITE to the CC and updates its cache line's state to valid within the shim (if it is not already). When the CC receives this WRITE, it writes its data into its own cache and forwards the WRITE to each cluster that is registered as a sharer of the updated cache line. The cluster whose shim sent the original WRITE message is added as a sharer. When remote sharers receive the WRITE, they propagate it within their clusters (Table I).

Cluster reads. Clusters may always service reads with data they have cached locally. On a read miss at the LLC, the shim sends a RREQ to the CC and does not service local cluster instructions until it receives back a RRESP. Upon receiving

[4]Note that RC's $\mathtt{W}_{\mathtt{rel}}$ operations have a slightly different semantics compared to C11's $\mathtt{W}_{\mathtt{rel}}$ operations, but are similar in spirit.



(a) Concurrent writes violate coherence. (b) SC writes violate the SC axiom.

Fig. 8: Motivating refinements to the MEMGLUE$_{\text{O}}$ protocol.

the RRESP, the shim services the cluster read by supplying this data to its LLC and updating its state in the shim to valid.

Cluster fences. The shim sends a FREQ to the CC and stalls handling all cluster requests. The CC responds with a FRESP.

B. *Refining the Protocol*

We refine the §IV-A protocol in two ways to maintain C11.

1) Timestamps: Recall that the C11 Coherence axiom enforces SC-per-location (§II-B), which requires that all threads agree on a total order for same-address memory operations. The simple MEMGLUE protocol described in the previous section violates this notion.

Consider the example in Fig. 8a. Note the omission of instruction strengths; the problematic behavior of this example is present under any mapping of instructions to strengths. Without loss of generality, suppose the WRITEs arrive in ascending order at the CC, and x is initially 0. To maintain SC-per-location, as required by Coherence, both shims must observe these same-address writes in the same order. However, under our current simple protocol, Shim 1 observes the write order to be $\mathtt{1,2,3}$, while Shim 2 observes $\mathtt{3,1,2}$, because each shim indiscriminately overwrites its local data with the forwarded WRITE updates it receives. MEMGLUE$_{\text{O}}$ corrects this behavior with *timestamps*.

Each cache line's metadata in the shims and CC is extended with a timestamp (TS) (Fig. 10). Each time the shims or CC process a cluster write hit / write-back (shims) or MEMGLUE WRITE (shims or CC), they increment the TS they track for the target cache line. On a write miss, a shim *synchronizes* its TS for the write's cache line with the CC, by acquiring the CC's TS and setting its local TS equal to it. Any WRITE sent from the CC to a shim is tagged with the CC's TS. When the shims receive a WRITE, they perform a *timestamp check* to determine whether to propagate the WRITE's data within their clusters.

Definition IV.1. *For* WRITE w*, shim* S*, and address* a*, the* timestamp check *determines whether* w*'s timestamp exceeds the shim's timestamp at* a*, i.e.* $\mathtt{w.TS} > \mathtt{shim[a].TS}$.

The WRITE is propagated within the local cluster *only if* the timestamp check passes; otherwise, its data is stale and must be discarded. In either case, the shim increments its local timestamp. Now, in Fig. 8a, when $\mathtt{Wx} = \mathtt{1}$ arrives to Shim 2, its timestamp and the shim's timestamp for x will be both be 1. Thus, the timestamp check will fail and the shim's timestamp will be incremented to 2 without overwriting the value 3. The same happens when $\texttt{Wx} = 2$ with timestamp 2 arrives at Shim 2. However, when $\texttt{Wx} = 3$ with timestamp 3 arrives at Shim 1, the timestamp check passes, and 3 is written. This means both shims will have data 3 and TS 3 in their caches at the end of the exchange. Using these timestamps, we prove that MEMGLUE$_O$ upholds SC-per-location:

Theorem IV.1. *For all addresses* $\texttt{a}$, $\exists \prec_{\texttt{a}}$ *a total order on all writes to* $\texttt{a}$*, such that for all shims* $\texttt{S}$*, and any pair of reads* $\texttt{R} \rightarrow_{\texttt{sb}} \texttt{R}'$ *on S which read values* $\texttt{w}$ *and* $\texttt{w}'$, $\texttt{w} \preceq_{\texttt{a}} \texttt{w}'$.

2) SC Writes: Consider the motivating example in Fig. 8b. Suppose both shims initially cache x and y, both with value 0. Given the current simple protocol, both WRITEs are sent to the CC, and then both reads immediately read the cached data (0) before the remote WRITEs arrive. The outcome would therefore be observable, despite being forbidden by C11's SC axiom. To address this, MEMGLUE$_O$ requires that a shim stop servicing local cluster requests after outputting an SC WRITE until it has received a WRITE_ACK back from the CC. This requirement forces prior SC WRITEs (that reached the CC before the shim's SC WRITE) to propagate to the shim before it may service future instructions. Doing so ensures that SC reads observe a total order for SC writes, as C11 requires (§II-B).

C. System-wide Proof of MEMGLUE$_O$

For MEMGLUE to uphold C11, the program executions observable in MEMGLUE must be a subset of those allowed by C11 (i.e., MEMGLUE $\subseteq$ C11). In this section, we sketch three out of the four proofs that we conduct to verify that MEMGLUE$_O$ upholds the C11 axioms for all programs. All four proofs—one corresponding to each C11 axiom (§II-B)—can be found in our open-source repository [1].

Each proof proceeds as follows. First, we assume the existence of an axiom-violating program execution. Then, we establish an order $\prec_{CC}$ in which messages must have hit the CC for this execution to have been observable in MEMGLUE$_O$ (e.g., $\texttt{m0} \prec_{CC} \texttt{m1}$ means m0 hits the CC before m1). Then, we derive a contradiction ($\Rightarrow\!\Leftarrow$) that $\prec_{CC}$ must contain a cycle, proving that the execution is not observable in MEMGLUE$_O$.

Coherence (§II-B): $\nexists \texttt{I1}, \texttt{I2}.(\texttt{I1}, \texttt{I2}) \in \texttt{hb} \wedge (\texttt{I2}, \texttt{I1}) \in \texttt{eco}$

Pf (sketch). We assume for sake of contradiction that such instructions I1 and I2 exist. We use the orderedness of the network and Thm. IV.1 to prove that instructions related by hb hit the CC in hb order. We then prove that eco-related instructions must hit the CC in eco-order by casing on each eco edge type. Then, $\texttt{I1} \prec_{CC} \texttt{I2}$ because $(\texttt{I1},\texttt{I2}) \in \texttt{hb}$, but also $\texttt{I2} \prec_{CC} \texttt{I1}$ because $(\texttt{I2},\texttt{I1}) \in \texttt{eco}$. $\Rightarrow\!\Leftarrow$ $\square$

SC Axiom (§II-B): acyclic (psc)

Pf (sketch): Recall that psc orders SC operations with respect to one another (Fig. 2). Assume a cycle of psc edges exists in some program execution. We first prove that any psc cycle must contain at least one write or fence. Then we prove that all WRITEs and FREQs generated in this cycle must hit the CC in psc-order, meaning that $\prec_{CC}$ contains a cycle. $\Rightarrow\!\Leftarrow$ $\square$

x86 instruction	Generated by (C11)	Translated to (MemGlue)
MOV (from memory)	L_{RLX},L_{ACQ},L_{SC}	L_{SC}
MOV (into memory)	S_{RLX},S_{REL},S_{SC}	S_{SC}
MFENCE	F_{SC}	F_{SC}

(a) TSO.

ARM instruction	Generated by (C11)	Translated to (MemGlue)
LDR	L_{RLX}	L_{RLX}
LDA	L_{ACQ}, L_{SC}	L_{SC}
STR	S_{RLX}	S_{RLX}
STL	S_{REL},S_{SC}	S_{SC}
DMB ISH LD	F_{ACQ}	F_{SC}
DMB ISH	F_{REL}, F_{ACQREL},F_{SC}	F_{SC}

(b) ARMV8.

Fig. 9: C11 compiler mappings, and MEMGLUE reverse compiler mappings for loads (L), stores (S), and fences (F). Recall that MEMGLUE does not yet support non-SC fences.

No-Thin-Air (§II-B): acyclic (sb|rf)

Pf (sketch). Assume a (sb|rf) cycle exists in some program. Then prove: $\forall \texttt{I}_1, \texttt{I}_2.\ (\texttt{I}_1 \rightarrow_{\texttt{sb}} \texttt{I}_2 \vee \texttt{I}_1 \rightarrow_{\texttt{rf}} \texttt{I}_2) \implies \texttt{I}_1 \prec_{CC} \texttt{I}_2$. A cycle in (sb|rf) thus implies a cycle in $\prec_{CC}$. $\Rightarrow\!\Leftarrow$ $\square$

D. Per-Cluster Proofs

§IV-C presents a proof that MEMGLUE$_O$ upholds correct C11 instruction orderings; it remains to be shown that shims correctly translate local coherence protocol messages to C11-style MEMGLUE messages. We design shim translation units assuming clusters run (correctly) compiled C11 code.

1) Translation Scheme: In a nutshell, shims observe cluster coherence protocol messages, determine the ISA instruction(s) that generate these messages, identify the *strongest* C11 operations that generate these instructions [69], and output the matching MEMGLUE messages. That is, we design shims to effectively invert (verified) compiler mappings from C11 to a cluster's target ISA MCM,[5] as illustrated in Fig. 9 [69].

Our translation strategy clearly enforces MEMGLUE $\subseteq$ C11 in the absence of compiler optimizations. However, compilers may perform *legal* optimizations, which guarantee ISA $\subseteq$ C11 [6], [16], [17], [29], [28], [77]. We sketch a proof by contradiction that in the presence of such optimizations, MEMGLUE $\subseteq$ C11 still holds. Suppose that a program is compiled to p (unoptimized) and p_{opt} (legally optimized). Shims enforce MEMGLUE $\subseteq$ C11 iff three conditions hold.

Condition 1: Instructions in p_{opt} are at least as strong as their p analogs. That is, under the mapping $orig : inst \rightarrow inst$ from instructions in p to their counterparts in p_{opt}, $\forall i, i'.(i, i') \in orig \implies stren(i') \not\prec stren(i)$ (recall from §II-B that RLX $\prec$ REL/ACQ $\prec$ ACQREL $\prec$ SC). Only a compiler optimization that relaxes the strengths of instructions in p to produce p_{opt} can violate the correctness condition above. But, such a relaxation would also violate ISA $\subseteq$ C11, contradicting our assumption on legal compiler optimizations. $\Rightarrow\!\Leftarrow$ $\square$

Condition 2: Source-to-source instruction reorderings, which happen at the C11 level before lowering to machine

[5]In the case of ISAs whose MCMs are not C11-compatible, MEMGLUE can translate all instructions to SC, but cannot exploit their relaxed MCMs.

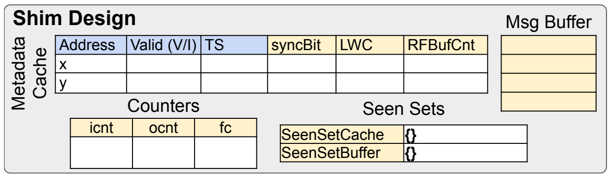


Fig. 10: MEMGLUE shim design. Blue components are those of $\text{MEMGLUE}_{\text{O}}$, yellow are those added by $\text{MEMGLUE}_{\text{U}}$.

code, may not violate C11. Legal compilers perform only source-to-source transformations which uphold C11 [77]. □

Condition 3: $\forall \texttt{I1}, \texttt{I2}. (\texttt{I1}, \texttt{I2}) \in \texttt{sb}_p \wedge (\texttt{I2}, \texttt{I1}) \in \texttt{sb}_{popt} \implies (\texttt{I1}, \texttt{I2}) \notin \texttt{sb}_{\texttt{cause}}$ (we use $\texttt{sb}_{\texttt{cause}}$ to capture those thread-local orderings that must be globally enforced by the ISA MCM). That is, instructions may only get reordered in the optimized program if such a reordering is permitted by the ISA MCM. Any compiler violating this condition may produce code that violates its ISA MCM. However, if compilers violate their MCM, they lose any provable guarantee that ISA $\subseteq$ C11. Thus, such reorderings would not be legal. $\Rightarrow\!\Leftarrow$ □

V. UNORDERED MEMGLUE CONSISTENCY PROTOCOL

MEMGLUE is intended to be implemented over a network with no ordering guarantees, yielding Unordered MEMGLUE ($\text{MEMGLUE}_{\text{U}}$). As a motivating example, recall the programs from Fig. 5; consider what happens if the two writes on Core 1 are sent to Core 2 and arrive out-of-order. In Fig. 5b, $\text{MEMGLUE}_{\text{U}}$ should not reconstruct the original ordering because the writes are allowed to be read out-of-order. However, in Fig. 5a, this reordering should not be visible due to the REL-ACQ synchronization between the cores. $\text{MEMGLUE}_{\text{U}}$ must track enough metadata to distinguish cases like these and reconstruct the proper ordering of messages when necessary.

A. Reorderings Allowed in $\text{MEMGLUE}_{\text{U}}$

In this section, we distinguish between messages *arriving* versus *accepting* at a destination. A message *arrives* when it reaches its destination after being sent through the network. A message *accepts* (after arriving) once its destination is allowed, per $\text{MEMGLUE}_{\text{U}}$'s state transition rules, to *process* it (e.g., update state, send response messages). A message *arrives early* if it reaches its destination before all prior messages from the same sender have been accepted at the destination. A message *accepts early* if it arrives early, and then is accepted before all prior messages from the same sender to the same destination have been accepted. Any $\text{MEMGLUE}_{\text{U}}$ message may arrive early; only some may accept early.

$\text{MEMGLUE}_{\text{U}}$ permits the following optimizations:

1) `RLX` reads from a cluster may read from `WRITE`s that have arrived early to the shims.
2) `RLX` `WRITE`s and `RRESP`s may accept early.
3) `REL` `WRITE`s and `ACQ` `RRESP`s may accept early.

Each reordering is subject to certain constraints (§V-B).

$\text{MEMGLUE}_{\text{U}}$ tracks additional metadata, shown in Fig. 10. To reconstruct the order in which messages were originally sent to them from each sender, the shims and CC maintain a set of *message counters*: an `icnt` per (incoming) message source and an `ocnt` per (outgoing) message destination. These track the number of messages received at and sent by each shim/CC, respectively. The shims only have one source and destination for all messages, the CC, and thus only have one `icnt` and `ocnt`. A message arrives early if its `cnt` (i.e., the `ocnt` of its sender at the time it was sent) is more than one greater than the destination's `icnt` for its sender. When a message arrives early, but cannot accept early, it is queued in a *message buffer*. Messages are removed from the buffer either when enough prior messages have accepted such that the buffered messages may accept early, or when all messages from the same sender with a lower `cnt` have accepted. A counter `RFBufCnt` is tracked per cache line to maintain SC-per-location under the first optimization (see our open-source repository [1] for details). $\text{MEMGLUE}_{\text{U}}$ also tracks `write_ids`, `seen_ids`, `seen_sets`, and `fence_counters` (§V-B2), as well as `local_write_counters` (§V-B1).

B. $\text{MEMGLUE}_{\text{U}}$ *Protocol*

$\text{MEMGLUE}_{\text{U}}$ enables significantly more reordering of protocol messages than $\text{MEMGLUE}_{\text{O}}$. We describe how it retains SC-per-location (§IV-B) and `hb` orderings (§II-B) below.

1) SC-per-location: Same-address write updates may arrive to the shims out-of-order, potentially causing a stale write to pass the timestamp check (Def. IV.1). This scenario would occur in the execution in Fig. 8a if the write updates of $\texttt{x} = \texttt{1}$ and $\texttt{x} = \texttt{2}$ arrive out-of-order to Shim 2. Therefore, in $\text{MEMGLUE}_{\text{U}}$, the shims must accept all same-address write updates in order. To this end, the shims track a `local_write_counter` (`LWC`) per address, and the CC tracks a `LWC` per address, per shim. The `LWC`s function similarly to the `icnts`/`ocnts` and ensure that same-address write updates accept in order. With this ordering guarantee, the normal timestamp check (§IV-B) may be used in $\text{MEMGLUE}_{\text{U}}$.

2) Happens-Before Orderings: To maintain the `hb` relation, instructions related by `sw` must correctly enforce orderings induced by release-acquire synchronization, as described in §II-B. This is difficult in $\text{MEMGLUE}_{\text{U}}$, as `REL` `WRITE`s and `ACQ` `RRESP`s can arrive to the shims out-of-order with respect to write updates that happened before them. As an example, consider Fig. 11. Instruction `1` (`I1`) synchronizes with `I2`, meaning $\texttt{I1} \rightarrow_{\text{hb}} \texttt{I3}$. However, `I1` and `I3`'s write updates may arrive out-of-order to Shim 3. This reordering would render the forbidden outcome in Fig. 11 observable, so it should not be allowed. However, some reordering of `REL` and `ACQ` messages should be allowed, if the shim has already seen[6] the writes that are required in order to maintain `sw`-induced orderings.

To determine whether `REL`/`ACQ` reordering is allowed, we add (1) unique *write ids* per write, assigned at the CC, (2) a *seen id* per (`REL`/`ACQ`) message, to track the highest `write_id` a shim must see before accepting the message, and (3) two

[6]"Seen" is formally defined the proof [1]. Intuitively, a core has "seen" a write once no reads on that core can read a from another write older than it.

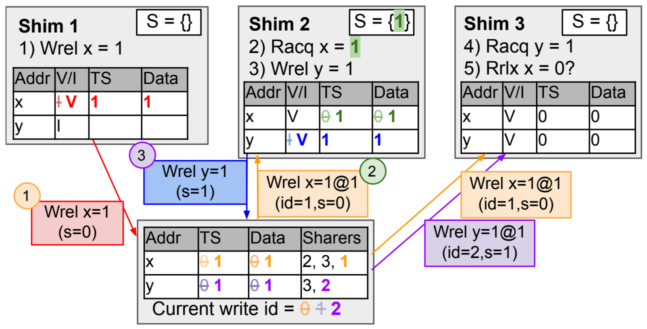


Fig. 11: Example of MemGlue$_U$ forbidding an execution of a litmus test with REL-ACQ synchronization. Events unfold in rainbow order. Shim and CC structures have been simplified.

seen sets per shim, to track what writes each shim has seen. For simplicity, we elide details of the distinction between each `seen_set`; details can be found in our open-source repository [1]. When REL WRITEs are sent to the CC, they carry with them the highest `write_id` that has previously arrived at the sending shim. In Fig. 11, for example, there are no writes in Shim 1's `seen_set` (S) when I1 performs, so I1 carries `seen_id` $= 0$ with its REL WRITE update. This write gets assigned `write_id` $= 1$ at the CC and is then sent to Shim 2 as an update. When this update arrives and is accepted at Shim 2, `write_id` $= 1$ is added to Shim 2's `seen_set`, signifying that any remote instruction that synchronizes-with any later instruction at Shim 2 must see I1. So, when I3 performs, its update to the CC carries `seen_id` $= 1$, and the forwarded update to Shim 3 carries this `seen_id` as well. Crucially, Shim 3 *cannot* accept this update until `write_id` $= 1$ is present in Shim 3's `seen_set`. If Shim 3 is not registered as a sharer of the cache line associated with the update's `seen_id` (i.e., it will never be forwarded an update with `write_id` $= 1$), then it will not be able to accept the update early. Hence, MemGlue$_U$ will not allow I5 to read 0, which would violate Coherence. This "seen" logic prevents write updates from arriving before hb-prior messages, ensuring MemGlue$_U$ honors sw.

Fences may also be related by sw (§II-B). As such, messages must not get reordered across them: all writes that happened before them must be seen by any read or fence that synchronizes with them. While trivial to achieve in MemGlue$_O$ due to the orderedness of the network, in MemGlue$_U$ we must add additional *fence counters* (`fcs`) to preserve these orderings.

When the CC receives a FREQ, it forwards it to all other shims. The CC's `fcs` count how many FREQs are sent to each shim, and a shim's `fc` counts how many FREQs it has received. Each CC message to the shims is tagged with the CC's `fc` for that shim; when a message `msg` arrives to a shim, if `msg.fc` $\neq$ `shim.fc`, then `msg` has arrived before a prior fence. The shim buffers `msg` until it has seen `msg.fc` total FREQs.

C. System-wide Proof of MemGlue$_U$

Coherence (§II-B): This proof proceeds exactly as the original proof (§IV-C), but with MemGlue$_U$'s ordering relaxations factored in. For instance, to reason about orderings involving ACQ reads and REL writes, we introduce the "seen" relation in the proof, which is inspired by the intuitive definition we presented in §V-B2. We prove that if I1 $\rightarrow_{hb}$ I2, then I2 "saw" I1, and that if I2 $\rightarrow_{eco}$ I1, then I1 "saw" I2. This inverse seen relation presents our contradiction.

SC Axiom (§II-B): Since SC instructions are always accepted in order in MemGlue$_U$, this proof is nearly identical to the ordered proof. However, we reason differently about fences; when a fence is involved in an sw edge, it is necessary to prove that instructions that happen before a release fence are seen by all instructions that happen after an acquire fence (§II-B). We prove this via fence counters.

VI. Verifying MemGlue's Correctness

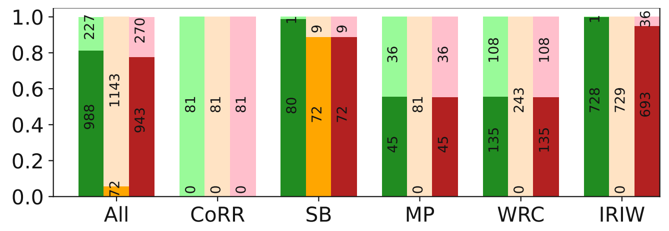
To verify MemGlue upholds C11, we (1) implement it in a model checker, and (2) complete a manual correctness proof.

A. Model Checking

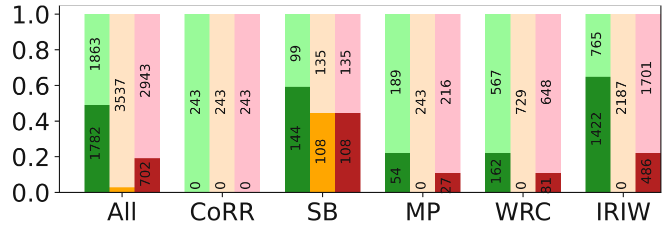
Murφ is an explicit-state model checker for concurrent systems commonly used to verify cache coherence protocols [27]. We first implement MemGlue$_O$ and MemGlue$_U$ in Murφ, and verify these implementations with respect to a test suite derived from the CoRR, SB, MP, WRC, and IRIW litmus tests [7]. The first suite of 1,215 tests features all variations of these litmus tests produced by assigning each instruction with each relevant C11 memory order (§III-A). The second suite of 3,645 tests is derived from the first by considering all possible placements of SC fences. For all tests, we treat clusters as black-boxes that emit MemGlue operations as defined in their assigned litmus test thread. Our goal is to verify the MemGlue protocol itself independent of shim translation.

We run each test through Murφ to determine its observability in MemGlue, and through the herd tool [8] using the RC11 model [45] (axiomatically defined in the cat language [8]) to determine its allowability in C11. Fig. 12a shows the results of running these tests with Murφ. For each implementation, no test forbidden by C11 is observable in either MemGlue variant—both uphold C11 with respect to the litmus tests. Also, MemGlue$_U$ allows most of the behavior that C11 does, meaning that MemGlue$_U$'s reordering optimizations are indeed leveraging the reordering behavior that is allowable by C11 (and thus the weak MCMs C11 accommodates). While not a performance study per se, this result suggests that the MemGlue protocol itself should not overly restrict heterogeneous shared memory performance.

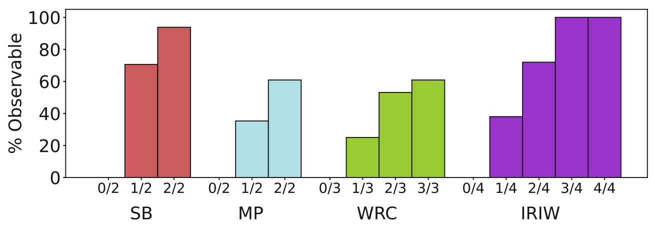
Next, we run a suite of 1,878 tests on MemGlue$_U$, in which we map all variations of the five tests across a set of "strong" and "weak" clusters. The "strong" clusters implement a standard MSI protocol locally [59], and we reverse-compile all instructions to SC to model a TSO cluster with a standard MSI coherence protocol (§IV-D1). The "weak" clusters are again black boxes in that they do not contain a local protocol, and reverse compilation could emit any combination of the atomics (modeling a cluster which maximally exploits the strengths offered by C11). The goal of these experiments is to



(a) Ordered (yellow) and Unordered (red) MEMGLUE results. Green columns show what is permitted in C11. Dark (light) colors are the portion of observable (unobservable) tests.



(b) Results of tests with all distributions of fences.



(c) Results of tests run on different fractions of weak (versus strong) cores.

Fig. 12: Litmus testing results.

demonstrate that MEMGLUE$_U$ permits more relaxed behavior as the clusters it unifies become weaker.

Fig. 12c shows that as more litmus test threads are mapped to weak clusters, more reordering is allowed by MEMGLUE$_U$.

B. *Proof*

Since our Murφ model checking results represent *bounded* proofs of MEMGLUE's correctness guarantees, we construct a manual proof that for any program, none of its C11-forbidden executions are observable in a MEMGLUE system (as sketched in §IV-C and §V-C). This is a particularly important result of this work—proving that a cache coherence protocol implements a particular MCM is notoriously difficult, even with protocols and MCMs that are significantly simpler than MEMGLUE and C11 [15]. This proof is also reusable across all MEMGLUE-enabled systems; only the shim-local proofs need to be re-done for each new cluster.

VII. RELATED WORK

Coherence Interfaces: Some works propose novel coherence interfaces to support fine-grained heterogeneous coherence.

Crossing Guard [62] provides a MESI-style coherence interface between a host CPU and accelerators. Beyond correctness, the main goal of Crossing Guard is to ensure safe and reliable interactions of untrusted accelerators with the host, by defending against unauthorized data access, deadlock, and denial of service attacks. However, to achieve some of these guarantees, Crossing Guard may require host coherence protocol changes.

Spandex [9] provides a richer coherence interface based on the DeNovo coherence protocol [24], with the primary goal of high-performance integration of heterogeneous devices with a wider range of coherence protocol demands. Spandex's device-side logic and integration logic are comparable to MEMGLUE's shims and CC, respectively. The authors discuss on how Spandex could be extended to account for inter-device MCM mismatches, but do not implement these extensions.

Instead of a coherence interface per se (like above or industrial approaches [5], [71], [73], [12], [31], [2]), MEMGLUE provides an MCM interface and adopts an update-based consistency protocol design to intercept and propagate relevant cluster operations, according to their ISA MCM requirements.

Consistency Protocols: HeteroGen [63], which synthesizes a consistency protocol for a particular set of heterogeneous clusters, is the first work to explicitly address MCM mismatches among clusters in heterogeneous coherence protocol design. §I discusses the trade-offs associated with this approach.

Follow-up work [34] presents a compositional operational model for defining *compound memory models*, which result from merging together per-cluster MCMs via a HeteroGen-style approach. The operational model can handle scoped and non-MCA cluster MCMs (unlike HeteroGen) by leveraging ordering relaxation in message propagation and predecessor tracking of memory operations—similar in spirit to MEMGLUE's unordered update propagation and seen sets, respectively. However, the model assumes that threads have a global knowledge of where instructions have propagated in order to maintain correct instruction orderings, challenging its transformation into a concrete implementation. MEMGLUE, in contrast, is designed to be implementable in hardware.

VIII. CONCLUSIONS

MEMGLUE is an update-based consistency protocol that facilitates cache-coherent shared memory among heterogeneous clusters with diverse MCMs. To do so, it equips each cluster with a hardware shim that translates relevant cluster coherence protocol messages to C11-style MEMGLUE messages, and then coordinates the exchange of MEMGLUE messages among shims. We prove that MEMGLUE upholds C11 with respect to several thousand litmus tests (using model checking) and for all programs (with a manual proof).

ACKNOWLEDGMENT

We thank Grigory Chirkov and the anonymous reviewers for their constructive comments and feedback. This work was supported by the National Science Foundation (NSF) under the Graduate Research Fellowship Program and award number CAREER CCF-2236855. Rachel would also like to acknowledge and celebrate her late father, Dr. Rance Cleaveland, for his guidance, encouragement, and support throughout this project and all of her research work. He is dearly missed by his family, his friends, and the academic community.

References

[1] https://github.com/rachelcleaveland/memglue-litmus-testing.

[2] Cache coherent interconnect for accelerators (ccix). https://www.ccixconsortium.com/. Accessed: 2023-08-14.

[3] Heteogeneous integration roadmap 2021 edition. https://eps.ieee.org/technology/heterogeneous-integration-roadmap/2021-edition.html. Accessed: 2023-11-09.

[4] memory_order. https://en.cppreference.com/w/c/atomic/memory_order. Accessed: 2023-07-11.

[5] Nvidia grace hopper superchip architecture. Technical report, Nvidia Corporation, Santa Clara, CA, 2022.

[6] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007.

[7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. *Proceedings of the* 17^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[8] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014.

[9] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[10] Arm. Architecture reference manual, Armv7-A and Armv7-R edition, 2008.

[11] Arm. Arm architecture reference manual, Armv8, for Armv8-A architecture profile, 2013.

[12] Arm. Amba chi architecture specification, 2024. Accessed 31 July 2024.

[13] Russell R Atkinson and Edward M McCreight. The dragon processor. *ACM SIGOPS Operating Systems Review*, 1987.

[14] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, 2002.

[15] Christopher J. Banks, Marco Elver, Ruth Hoffmann, Susmit Sarkar, Paul Jackson, and Vijay Nagarajan. Verification of a lazy cache coherence protocol against a weak memory model. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017.

[16] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[17] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[18] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. *43rd Symposium on Principles of Programming Languages (POPL)*, 2016.

[19] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[20] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, 2014.

[21] Sebastian Burckhardt, Rajeev Alur, and Milo MK Martin. Verifying safety of a token coherence implementation by parametric compositional refinement. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2005.

[22] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[23] Liqun Cheng and John B Carter. Extending cc-numa systems to support write update optimizations. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[24] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[25] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 2020.

[26] Christina Delimitrou and Christos Kozyrakis. Quality-of-service-aware scheduling in heterogeneous data centers with paragon. *IEEE Micro*, 2014.

[27] David L Dill. The mur ϕ verification system. In *Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8*, 1996.

[28] Johannes Doerfert and Hal Finkel. Compiler optimizations for openmp. In *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings 14*, 2018.

[29] Johannes Doerfert and Hal Finkel. Compiler optimizations for parallel programs. In *International Workshop on Languages and Compilers for Parallel Computing*, 2018.

[30] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. Verifying c11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.

[31] HSA Foundation. Heterogeneous system architecture: A technical review, 2012. Accessed 31 July 2024.

[32] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[33] David B Glasco, Bruce A Delagi, and Michael J Flynn. Update-based cache coherence protocols for scalable shared-memory multiprocessors. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994.

[34] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. Compound memory models. *Proceedings of the ACM on Programming Languages*, 2023.

[35] Håkan Grahn, Per Stenström, and Michel Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 1995.

[36] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 2019.

[37] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *Commun. ACM*, 2021.

[38] IBM. Power ISA version 2.07, 2013.

[39] ISO/IEC. Information technology – programming languages – C. International standard 9899:2011, 2011.

[40] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[41] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *ACM SIGPLAN Notices*, 2017.

[42] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE micro*, 2011.

[43] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, 1992.

[44] Leonidas I Kontothanassis, Michael L Scott, and Ricardo Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, 1995.

[45] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. https://plv.mpi-sws.org/scfix/.

[46] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[47] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 1979.

[48] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[49] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *Proceedings of the* 47th *International Symposium on Microarchitecture (MICRO)*, 2014.

[50] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. *Proceedings of the* 24th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[51] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. *Proceedings of the* 21st *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[52] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. *42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[53] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated memory consistency proofs for microarchitectural specifications. *Proceedings of the* 51st *International Symposium on Microarchitecture (MICRO)*, 2018.

[54] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. *Proceedings of the* 50th *International Symposium on Microarchitecture (MICRO)*, 2017.

[55] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. *Proceedings of the* 48th *International Symposium on Microarchitecture (MICRO)*, 2015.

[56] Jeremy Manson, William Pugh, and Sarita V Adve. The java memory model. *ACM SIGPLAN Notices*, 2005.

[57] A. Meixner and D.J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

[58] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. Finepack: Transparently improving the efficiency of fine-grained transfers in multi-gpu systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[59] Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[60] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for c/c++ 11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[61] NVIDIA. Parallel thread execution ISA version 6.0., 2017. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[62] Lena E. Olson, Mark D. Hill, and David A. Wood. Crossing guard: Mediating host-accelerator coherence interactions. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[63] Nicolai Oswald, Vijay Nagarajan, Daniel J Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. Heterogen: Automatic synthesis of heterogeneous cache coherence protocols. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[64] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, 2009.

[65] Fong Pong, Andreas Nowatzyk, Gunes Aybay, and Michel Dubois. Verifying distributed directory-based cache coherence protocols: S3. mp, a case study. In *EURO-PAR'95 Parallel Processing: First International EURO-PAR Conference Stockholm, Sweden, August 29–31, 1995 Proceedings 1*, 1995.

[66] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[67] Oxford University Press. Oxford advanced learner's dictionary, 2024. https://www.oxfordlearnersdictionaries.com/.

[68] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages*, 2017.

[69] Peter Sewell. C/c++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. Accessed: 2023-07-11.

[70] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 2010.

[71] Debendra Das Sharma and Siamak Tavallaei. Compute express link 2.0 white paper. *CXL. Retrieved October*, 31:2021, 2020.

[72] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 1990.

[73] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 2015.

[74] Charles P Thacker and Lawrence C Stewart. Firefly: a multiprocessor workstation. *ACM SIGARCH Computer Architecture News*, 1987.

[75] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. *Proceedings of the* 22nd *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[76] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[77] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[78] Andrew Waterman and Krste Asanović. The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20190608-base-ratified. Technical report, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, June 2019.

[79] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. Cohort: Software-oriented acceleration for heterogeneous socs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[80] Will Deacon. Formalising the armv8 memory consistency model. https://www.csm.ornl.gov/workshops/openshmem2018/presentations/mm-openshmem2018.pdf, August 2018.

[81] Andrew W. Wilson and Richard P. LaRowe. Hiding shared memory reference latency on the galactica net distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 1992.

[82] Meng Zhang, Alvin R Lebeck, and Daniel J Sorin. Fractal coherence: Scalably verifiable cache coherence. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[83] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

[84] Zhen Zhuang, Bei Yu, Kai-Yuan Chao, and Tsung-Yi Ho. Multi-package co-design for chiplet integration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022.

Translating Pseudo-Boolean Proofs into Boolean Clausal Proofs

Karthik V. Nukala, Soumyaditya Choudhuri, Randal E. Bryant, Marijn J. H. Heule
Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Email: {kvn, soumyadc}@andrew.cmu.edu, {rebryant, marijn}@cmu.edu

***Abstract*—Clausal proofs, particularly those based on the deletion resolution asymmetric tautology (DRAT) proof system, are widely used by Boolean satisfiability solvers for expressing proofs of unsatisfiability. Their success stems from their simplicity and scalability. When solvers go beyond pure propositional reasoning, however, generating clausal proofs becomes more difficult. Solvers that employ pseudo-Boolean reasoning, including cutting-planes operations, can express proofs in the VeriPB proof system, but its adoption is not widespread.**

We introduce PBIP (Pseudo-Boolean Implication Proof), a framework that provides an intermediate representation between VeriPB and clausal proofs. We also introduce a toolchain comprising 1) a VeriPB-to-PBIP translator that performs proof trimming and optimization, and 2) a PBIP-to-LRAT translator that makes use of proof-generating operations on ordered binary decision diagrams (BDDs) to generate clausal proofs in LRAT format, a variant of the DRAT that allows efficient checking.

We demonstrate the viability of our approach, the effectiveness of our trimming, and the performance of our clausal proof generator on a set of native PB benchmarks and compare our approach to direct checking of VeriPB proofs.

I. Introduction

Boolean satisfiability (SAT) solvers underlie a large portion of automated reasoning tools such as theorem provers, satisfiability modulo theory (SMT) solvers, and model checkers. Given the safety-critical application domains of these tools, correctness of the underlying solver is of utmost importance. Creating a formally verified solver (using an interactive theorem prover, for example) would severely compromise the ability to optimize and rapidly evolve the program. Most satisfiability solvers take as input formulas expressed in conjunctive normal form (CNF). These formulas consist of a conjunction of *clauses*, each of which is a disjunction of *literals*, where each literal is a Boolean variable or its complement.

An alternative to a formally verified solver is to have the solver generate a proof certificate for each execution. When this certificate is successfully checked by a verified proof checker, the result is guaranteed to be correct. The deletion resolution asymmetric tautology (DRAT) proof system [1] has become the standard for modern SAT solvers and is widely used by entries in the annual SAT competition [2]. DRAT is notably a clausal proof system: a proof consists of a sequence of clauses where each clause preserves the satisfiability of the preceding clauses. A proof of unsatisfiability terminates with the addition of the empty clause. Clausal proofs are of particular interest because they are simple (resulting in successful efforts to write verified proof checkers in interactive theorem provers such as ACL2 [3], Coq [4], and CakeML [5]) as well as scalable (being able to check proofs two petabytes in size [6]).

We consider pseudo-Boolean (PB) reasoning, chosen for its status as a bridge between propositional satisfiability and higher-level "beyond Boolean" reasoning. Also known as 0/1 integer linear programming, PB reasoning has been a fertile area of research since the 1950s. It has been one of longstanding multidisciplinary interest, with problems in operations research [7], combinatorics [8], economics [9], and VLSI design [10] (among others) benefiting from expressive encodings as pseudo-Boolean constraints. By virtue of these encodings, PB solvers can exploit richer structure and reason in a way that would be difficult for native SAT solvers to do. Notable PB solvers include PBS [11], Galena [12], Pueblo [13], and RoundingSAT [14]. Having a way to express and check PB proofs of unsatisfiability would enhance the level of trust users could place in these solvers.

The VeriPB proof framework [15]–[17] supports both the cutting planes (CP) proof system, viewing pseudo-Boolean constraints as linear constraints over 0/1-valued variables, and implication-based reasoning, viewing the constraints as Boolean formulas. With cutting planes, new constraints can be generated by summing two constraints or by scaling a single constraint by either multiplication or division. With implication-based reasoning, a new constraint can be added when it is shown to be implied by previous constraints via reverse unit propagation (RUP). Although a RUP-based implication can be translated into a sequence of cutting planes steps, RUP more directly captures the logical inferences made by some tools.

This paper describes a series of tools that can transform a VeriPB proof into a clausal proof in extended resolution [18], a proof system that lies within the DRAT proof framework. The generated proof is expressed in LRAT format, a variant of DRAT for which a variety of proof checkers have been developed, including ones that have been formally verified.

The key to our method is to represent pseudo-Boolean constraints as ordered binary decision diagrams (BDDs) [19], and to use a proof-generating BDD package to generate clausal proof steps justifying each of its operations [20]. The BDD representation of a pseudo-Boolean constraint over n variables and with maximum coefficient a will have at most $a \cdot n$ nodes,

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_23

and so we can say that the generated clausal proofs will be of *pseudo-polynomial* complexity relative to the VeriPB proofs. That is, the proofs will be polynomial in the values of the coefficients, but this can be exponential in the number of bits required to represent these values. In practice, many PB proofs involve only small coefficients, and so the expansion will be polynomial.

Some contributions of this work include:

- The ability to translate proofs in the relatively new and unfamiliar VeriPB framework into a more established clausal framework.
- The ability to directly compare the sizes of proofs generated using different approaches to logical reasoning.
- Methods to optimize VeriPB proof sizes and checking times by adapting some of the trimming and hint generation methods used in clausal proofs.
- Experimental results relating the sizes of the clausal proofs generated by several tools (including ours) to PB proofs. These serve to quantify the advantage of proof frameworks based on this higher level of reasoning. We show that the clausal proofs scale polynomially, relative to VeriPB, but their larger sizes pose challenges for more difficult benchmark problems.

We note several limitations to our work:

- The soundness of our toolchain relies on a program that generates CNF representations of the constraints comprising the input pseudo-Boolean formula. Although the current program is simple and has been thoroughly tested, it would be preferable to have one that has been formally verified.
- Our toolchain does not support the full suite of VeriPB proof rules. Most significantly, it cannot handle the two strengthening rules that enable symmetry reductions in VeriPB proofs [21].

We discuss these in Section VII.

II. Related Work

Recently, the CakePB [22] proof checker has been developed to enable formally verified checking of VeriPB proofs. It operates by first converting the original VeriPB proof into one in the VeriPB kernel format, where application of the RUP proof rule is expanded into a sequence of cutting-planes steps [23]. CakePB has the advantage that it can reason about operations on pseudo-Boolean constraints directly, rather than on their BDD representations. We compare the performance of our toolchain to one based on CakePB in Section VI. Our results show the effectiveness of proof trimming and motivate its addition to future versions of CakePB.

At first glance, having a translation from cutting-planes proofs into clausal proofs that only achieves pseudo-polynomial performance (in terms of the proof size) could seem to fall short of the theoretical optimum. In particular, W. J. Cook, et al. [24] sketch an algorithm for converting any cutting planes proof of unsatisfiability for CNF formula F into an extended resolution proof, such the number of steps in the extended resolution proof would be bounded by a polynomial function $p(n, m)$, with n equal to the total number of literals in F and m equal to the number of steps in the CP proof. That result is not directly comparable to ours, however:

- It assumes that the problem consists entirely of PB constraints encoding clauses, i.e., having only unit coefficients and a unit constant.
- The scale of the polynomial is not given in the paper, but it appears to be large. The presented translation requires converting the CP steps into many arithmetic operations on an encoded representation of the coefficients similar to a binary representation. [25].
- To our knowledge, the proposed algorithm has never actually been implemented and doing so would require a substantial effort.

By contrast, VeriPB allows the input formula to contain constraints with coefficients of arbitrary size. In addition, even when given a formula with small coefficients, the constraints in the VeriPB proof can have coefficients of arbitrary size. Cook's method would require encoding these with low-level arithmetic operations. This theoretical result is unlikely to translate into a practical method for proof generation.

III. Preliminaries

A. Pseudo-Boolean Formulas

We recommend the PhD thesis by Stephen Gocht [17] as a helpful introduction to pseudo-Boolean reasoning. A pseudo-Boolean constraint is a linear expression, viewing Boolean variables as ranging over integer values 0 and 1. That is, a constraint c has the form $a_1\ell_1 + a_2\ell_2 + \cdots + a_n\ell_n \;\#\; b$ where the coefficients a_i and the constant b are integers, and each *literal* ℓ_i equals either input variable x_i or its complement $\overline{x}_i$. For an *ordering* constraint, the relational operator $\#$ is $<$, $\leq$, $\geq$, or $>$. For an *equational* constraint, the relational operator is $=$. An equational constraint can also be represented as the conjunction of two ordering constraints having the same coefficients but one with relation $\leq$ and the other with $\geq$. We will mostly refer to *coefficient-normalized constraints* (CNCs) of the form

$$a_1\ell_1 + a_2\ell_2 + \cdots a_n\ell_n \quad \geq \quad b \tag{1}$$

where the coefficients and the constant are nonnegative integers and the relation is $\geq$.

An *assignment* ρ is a mapping from some subset of the variables in X to truth values 1 (true) and 0 (false). We can view an assignment as a set of literals $\rho = \{x_i \mid \rho(x_i) = 1\} \cup \{\overline{x}_i \mid \rho(x_i) = 0\}$. Assignment ρ is *total* when it assigns a value to every variable.

Constraint c denotes a Boolean function, written $[\![c]\!]$, mapping total assignments to truth values. Constraints c_1 and c_2 are said to be *equivalent* when $[\![c_1]\!] = [\![c_2]\!]$. Constraint c is said to be *infeasible* when $[\![c]\!] = \bot$, i.e., it always evaluates to 0. This occurs if and only if $\sum_{1 \leq i \leq n} a_i < b$. Constraint c is said to be *trivial* when $[\![c]\!] = \top$, i.e., it always evaluates to 1. This occurs if and only if $b = 0$.

As described in [17], the following are some properties of pseudo-Boolean constraints:

- A relational constraint with comparisons $<$, $\leq$, and $>$ can be converted to an equivalent CNC.
- An equational constraint can be converted into two CNCs.
- The logical negation of CNC c, written $\overline{c}$, can also be expressed as a CNC.
- Any coefficient a_i with $a_i > b$ in a CNC can be replaced with the coefficient b without changing the underlying Boolean function.

Some nomenclature regarding CNCs will prove useful. The *constraint literals* are those literals ℓ_i such that $a_i \neq 0$. A *cardinality constraint* has $a_i \in \{0, 1\}$ for $1 \leq i \leq n$. A cardinality constraint with $b = 1$ is referred to as a *clausal constraint*: at least one of the constraint literals must be assigned 1 to satisfy a constraint. It is logically equivalent to a clause in a conjunctive normal form (CNF) formula. A cardinality constraint with $b = \sum_{1 \leq i \leq n} a_i$ is referred to as a *conjunction*: all of the constraint literals must be assigned 1 to satisfy the constraint. A conjunction for which $a_i = 1$ for just a single value of i is referred to as a *unit* constraint: it is satisfied if and only if literal ℓ_i is assigned 1.

A pseudo-Boolean *formula* F is a set of pseudo-Boolean constraints. We say that F is *satisfiable* when there is some assignment ρ that satisfies all of the constraints in F, and *unsatisfiable* otherwise.

Although feasibility can readily be tested for individual CNCs, determining whether a set of constraints (even for set size 2) is satisfiable is intractable, unless $P = NP$. For example, the subset sum problem [26] can readily be translated into an equational constraint, and this can then be expressed as the conjunction of two CNCs.

Pseudo-Boolean optimization problems can be converted to decision problem by imposing a bound on the metric being optimized. For example, two runs of a PB solver suffice to prove that a graph has maximum clique size k. First, the solver is run with a cardinality constraint requiring clique size k. The generated solution can then be checked to make sure it is indeed a clique. Then the solver is run with proof generation enabled and with a cardinality constraint requiring clique size $k + 1$. The certificate of unsatisfiability completes the proof. Similar approaches can be used for other optimization problems [15].

B. (Reverse) Unit Propagation

We let $c|_\rho$ denote the CNC resulting when c is simplified according to partial assignment ρ. That is, assume c has the form of (1) and partition the indices i for $1 \leq i \leq n$ into three sets: I^+, consisting of those indices i such that $\ell_i \in \rho$, I^-, consisting of those indices i such that $\overline{\ell}_i \in \rho$, and I^X consisting of those indices i such that neither ℓ_i nor $\overline{\ell}_i$ is in ρ. With this, $c|_\rho$ can be written as $\sum_{1 \leq i \leq n} a'_i \geq b'$ with a'_i equal to a_i for $i \in I^X$ and equal to 0 otherwise, and with $b' = b - \sum_{i \in I^+} a_i$.

Literal ℓ_i is *unit propagated* by CNC c when the assignment $\rho = \{\overline{\ell}_i\}$ causes the constraint $c|_\rho$ to become infeasible. As the name implies, a unit-propagated literal ℓ_i then becomes a unit constraint. Observe that a single constraint can unit propagate multiple literals. For example, $4x_1 + 3\overline{x}_2 + x_3 \geq 6$ unit propagates both x_1 and $\overline{x}_2$. For CNC c, we let $\mathit{Unit}(c)$ denote the set of literals it unit propagates

Rather than simplifying a constraint c according to partial assignment ρ and then detecting unit propagations, we can combine these to detect the set of unit propagations for a constraint with respect to a partial assignment. That is, we define $\mathit{Unit}_\rho(c)$ to be $\mathit{Unit}(c|_\rho)$. These propagations can readily be detected by computing the *slack*, defined as $\mathit{Slack}_\rho(c) = \sum_{i \in I^X} a_i + \sum_{i \in I^+} a_i - b$, where I^X and I^+ are the sets of indices defined previously. $\mathit{Unit}_\rho(c)$ is then defined as $\{\ell_i \mid a_i > \mathit{Slack}_\rho(c)\}$. For example, the constraint $c \doteq 4x_1 + 3\overline{x}_2 + x_3 \geq 6$ has slack $4+3+1-6 = 2$ with respect to $\rho = \emptyset$. We can therefore compute $\mathit{Unit}_\rho(c) = \{x_1, \overline{x}_2\}$. Furthermore c will be infeasible for partial assignment ρ when $\mathit{Slack}_\rho(c) < 0$.

Given a set of constraints F, we can build up a partial assignment ρ by repeatedly performing unit propagation. That is, define the operation *Uprop* as $\mathit{Uprop}(\rho, c) = \rho \cup \mathit{Unit}_\rho(c)$. For initial assignment ρ, *unit propagation* on formula F is then the process of extending ρ by repeatedly computing $\rho \leftarrow \mathit{Uprop}(\rho, c)$ to all of the constraints $c \in F$ until no more propagations are possible.

Consider a formula F consisting a set of constraints $c_1, c_2, \ldots, c_m$. The *reverse unit propagation* (RUP) proof rule [15], [17] uses unit propagation to prove that *target constraint* c can be added to a formula while preserving its set of satisfying assignments. That is, any assignment that satisfies F also satisfies $F \wedge c$. A RUP addition justifies c by assuming $\overline{c}$ holds and showing, via a sequence of *RUP steps*, that this leads to a contradiction. It accumulates a partial assignment ρ based on unit propagations starting with the empty set. Each RUP step accumulates more assigned literals by performing a unit propagation of the form $\rho \leftarrow \mathit{Uprop}(\rho, d)$, where d is either c_j, a prior constraint, or $\overline{c}$, the negation of the target constraint. The final step causes a contradiction, where $d|_\rho$ is infeasible. Unlike with clauses, a single constraint, including the negated target, can be used for unit propagation on multiple RUP steps within a single RUP addition.

C. Trusted Binary Decision Diagrams

Trusted binary decision diagrams (TBDDs) [20] provide a method for generating clausal proofs when performing sequences of operations on Boolean functions represented as ordered binary decision diagrams (BDDs) [19]. TBDDs have been used to generate proofs of unsatisfiability for SAT solvers [27], proofs of satisfaction and refutation in QBF solvers [28], and for proofs of unsatisfiability for pseudo-Boolean constraints [29]. Proofs are generated directly in the LRAT format, making use of the support for extended resolution provided by the RAT proof system.

In the following, we write a clause consisting of literals $\ell_1, \ell_2, \ldots, \ell_k$ as $[\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k]$. A unit clause with literal ℓ is written as $[\ell]$.

The key idea is to introduce an extension variable u every time a BDD node $\mathbf{u}$ is created, with proof clauses defining the semantic relation between $\mathbf{u}$, the node variable x, and child nodes $\mathbf{u}_1$ and $\mathbf{u}_0$ [27], [30], [31]. Each step in the recursive algorithms to generate new BDDs generates a sequence of proof clauses justifying an inductive invariant about the operation being performed. For example, suppose the Apply algorithm [19] computes the conjunction of BDDs with root nodes $\mathbf{u}$ and $\mathbf{v}$ to derive a BDD with root node $\mathbf{w}$. Each recursive step of the operation performs the conjunction of argument nodes $\mathbf{u}'$ and $\mathbf{v}'$ to derive a node $\mathbf{w}'$. With TBDDs, this step also generates a sequence of proof steps concluding with the addition of clause $[\overline{u}' \vee \overline{v}' \vee w']$, justifying that $u' \wedge v' \Rightarrow w'$. The final step of the recursion then generates the clause justifying $u \wedge v \Rightarrow w$.

A *trusted* BDD $\dot{\mathbf{u}}$ is a BDD having root node $\mathbf{u}$ for which the unit clause $[u]$ has been added to the proof. That is, the BDD will evaluate to 1 for any assignment that satisfies the input formula. A proof of unsatisfiability concludes with the addition of the TBDD consisting of the leaf node L_0, representing $\bot$. This is encoded in the proof by the empty clause.

D. Cutting Planes

The cutting planes proof system defines rules to derive new constraints from existing ones, as is shown in Figure 1.

$$\text{DIV}\quad \frac{\sum_i a_i x_i \geq b}{\sum_i \frac{a_i}{k} x_i \geq \left\lceil \frac{b}{k} \right\rceil} \qquad \text{SAT}\quad \frac{\sum_i a_i x_i \geq b}{\sum_i \min(a_i, b) x_i \geq b}$$

$$\text{MUL}\quad \frac{\sum_i a_i x_i \geq b}{\sum_i k a_i x_i \geq kb} \qquad \text{ADD}\quad \frac{\sum_i a_i x_i \geq b \qquad \sum_i c_i x_i \geq d}{\sum_i (a_i + c_i) x_i \geq b + d}$$

Fig. 1. Cutting-Planes Proof Rules. For the division rule, each coefficient a_i must be divisible by k.

Notably, rules DIV, SAT, and MUL do not change the underlying Boolean function of the constraint. That is, for any constraint c, and $k \in \mathbb{N}^+$ (where each coefficient a_i in the division rule must be divisible by k):

$$[\![c]\!] = [\![\mathsf{DIV}(c, k)]\!] = [\![\mathsf{SAT}(c)]\!] = [\![\mathsf{MUL}(c, k)]\!]$$

Generating a constraint via the ADD rule, on the other hand, creates a constraint with a new underlying Boolean function, but it is implied by the conjunction of the Boolean functions for the arguments:

$$[\![c_1]\!] \wedge [\![c_2]\!] \Rightarrow [\![c_1 + c_2]\!]$$

IV. PBIP: Pseudo-Boolean Implication Proof

A Pseudo-Boolean Implication Proof (PBIP) provides a systematic way to prove that a PB formula F is unsatisfiable. The PBIP file format is described in Section IV-B. It is not intended to be a useful format on its own, but rather a bridge between a PB proof and its translations into a clausal proof.

A. PBIP Proof Structure

A PBIP proof is given as a sequence of constraints, referred to as the *proof sequence*:

$$c_1, c_2, \ldots, c_m, c_{m+1}, \ldots, c_t$$

such that the first m constraints are those of formula F, while each *added* constraint c_i (referred to as a *lemma*) for $i > m$ follows by implication from the preceding constraints. That is,

$$\bigwedge_{1 \leq j < i} [\![c_j]\!] \Rightarrow [\![c_i]\!] \tag{2}$$

The proof completes with the addition of an infeasible constraint for c_t. By the transitivity of implication, we have therefore proved that F is not satisfiable.

Constraints c_i with $i > m$, can be added in two different ways, corresponding to two different reasoning modes.

1) In *implication mode*, constraint c_i follows by implication from at most two prior constraints in the proof sequence. That is, some $H_i \subseteq \{c_1, c_2, \ldots, c_{i-1}\}$ with $|H_i| \leq 2$ satisfies
$$\bigwedge_{c_j \in H_i} [\![c_j]\!] \Rightarrow [\![c_i]\!] \tag{3}$$
Set H_i is referred to as the *hint* for proof step i.
To simplify the generation of PBIP proofs, the checker supports a *summation* rule of the form $\sum_{1 \leq i \leq k} d_i \Rightarrow c$, where each d_i is a constraint from a previous step. Checking this is performed by computing intermediate constraints as pairwise sums and proving that they satisfy implication.
2) In *RUP mode*, constraint c_i is justified by RUP addition. The hint specifies the RUP steps as a sequence $[d_1, m_1], [d_2, m_2], \ldots, [d_{k-1}, m_{k-1}], [d_k]$. Each d_j indicates either a previous constraint $c_{i'}$ for $i' < i$, or the negated target constraint $\overline{c}_i$. Each m_j indicates a unit-propagated literal. The final constraint, indicated by d_k should conflict with the accumulated assignment $\rho = \{m_1, m_2, \ldots, m_{k-1}\}$.

Unless $P = NP$, we cannot guarantee that a proof checker can validate even a single implication step of a PBIP proof in polynomial time. In particular, consider an equational constraint c encoding an instance of the subset sum problem, and let $c_{\leq}$ and $c_{\geq}$ denote its conversion into a pair of ordering constraints such that $[\![c]\!] = [\![c_{\leq}]\!] \wedge [\![c_{\geq}]\!]$. Consider a PBIP proof to add the constraint $\overline{c}_{\leq}$ having the $c_{\geq}$ as the only hint. Proving that $[\![c_{\geq}]\!] \Rightarrow [\![\overline{c}_{\leq}]\!]$, requires proving that $[\![c_{\leq}]\!] \wedge [\![c_{\geq}]\!] = \bot$, i.e., that c is unsatisfiable.

On the other hand, checking the correctness of a PBIP proof can be performed in *pseudo-polynomial* time using

BDDs, meaning that the complexity will be bounded by a polynomially sized formula over the numeric values of the integer parameters. In particular, a CNC over n variables and having a as the maximum of its coefficients a_i and the constant b will have a BDD representation with at most $a \cdot n$ nodes [32]. For an implication proof step where the added constraints and the hints all have coefficients and constants less than or equal to a, the number of BDD operations to validate the step will be $O(a^2 \cdot n)$ when there is a single hint and $O(a^3 \cdot n)$ when there are two hints. This complexity is polynomial in a, but it could be exponential in the size of a binary representation of a. The number of BDD operations for each unit propagation step in a RUP proof will be linear in the size of the BDD and therefore $O(a \cdot n)$.

B. PBIP File Format

A PBIP file describes a sequence of transformations on a set of pseudo-Boolean input constraints leading to an infeasible constraint. The file therefore describes an unsatisfiability proof for a PB constraint problem. The format assumes that each input constraint is encoded as a set of clauses in conjunctive normal form (CNF). The clauses for all of the constraints are provided as a file in the standard DIMACS format. The generation of this file is performed by a separate program PB-CNF, described in Section VI.

When in implication mode, each derived constraint must follow by implication from either one or two preceding constraints, referred to as the "antecedents". That is, for target constraint c and prior constraint c_1, and possibly c_2, we must have either $[\![c_1]\!] \Rightarrow [\![c]\!]$ or $[\![c_1]\!] \wedge [\![c_2]\!] \Rightarrow [\![c]\!]$. When in RUP mode, the constraint to be derived is set as a target, and a set of unit constraints is accumulated. Each RUP step then derives an additional unit constraint based on the previously derived unit constraints, as well as either the complement of the target constraint or some preceding constraint. The final step should then cause a contradiction, with the set of accumulated unit constraints falsifying the final constraint.

PBIP files build on the OPB format for describing PB constraints, as documented in [33].

There are five line types. The first four types define constraints that can be referenced by later lines. Constraints are numbered from 1, starting with the input constraints. File lines beginning with "`*`" are treated as comments.

1) Input lines begin with "`i`". This is followed by a constraint, expressed in OPB format, and terminated by "`;`". Then, a set of clause numbers from the CNF file is listed, separated by spaces and terminated with end-of-line. Forming the conjunction of these clauses and existentially quantifying any variables that are not listed in the PB formula should yield a Boolean function that is implied by the PB constraint.
2) Implication-mode assertion lines begin with "`a`". This is followed by a constraint, expressed in OPB format and terminated by "`;`". Then, either one or two constraint numbers is listed, separated by spaces and terminated with end-of-line.
3) RUP lines begin with "`u`". This is followed by a constraint, expressed in OPB format and terminated by "`;`". Then, a sequence of lists is given, where each list is of the form $[I\, \ell_1\, \ldots\, \ell_k]$, indicating that constraint number I will propagate additional units $\ell_1, \ldots, \ell_k$. I can either be the number of a previous constraint, or it can be that of the current constraint. The latter case is known as a "self reference", and its unit propagations should be based on the negation of the target constraint. The final list is of the form $[I]$, and the indicated constraint must be falsified by the accumulated set of literals. A list of the form $[I\, \ell_1\, \ldots\, \ell_k]$ with $k > 1$ indicates that multiple literals will be unit propagated. This notation is equivalent to listing the literals individually with the sequence $[I\, \ell_1]\, [I\, \ell_2] \cdots [I\, \ell_k]$.
4) Summation implication lines begin with "`s`". This is followed by a constraint c, expressed in OPB format and terminated by "`;`". Then, a set of constraint numbers is listed, separated by spaces and terminated with end-of-line. These numbers identify a set of prior constraints $c_1, c_2, \ldots, c_k$ satisfying:
$$\sum_{i=1}^{k} [\![c_i]\!] \quad \Longrightarrow \quad [\![c]\!]$$
This line avoids the need to expand a summation of k constraints into $k-1$ implication lines. Instead, the checker performs the summations and tests the final implication, using heuristics to optimize the order in which the argument constraints are summed.
5) Deletion lines begin with "`d`". This is followed by a list of constraint numbers. These constraints cannot be used as hints for the remainder of the proof.

For an unsatisfiability proof, the final constraint should be infeasible, e.g. $0 \geq 1$.

V. PBIP Trimming and Checking

Given a VeriPB proof, we must transform it into a PBIP proof. In doing so, we trim the proof to eliminate those steps that are not required for the final unsatisfiability result.

A. Hinting and Trimming Cutting-Planes and RUP Proofs

We present here a set of procedures that take a VeriPB proof of unsatisfiability (generated by a proof-logging solver such as RoundingSAT [34] or the Glasgow Subgraph/Clique Solvers [15], [35]) and perform a translation into PBIP. In addition, the resulting proof is *trimmed*, removing proof steps that do not lead to the final unsatisfiability result.

Our procedures support the following VeriPB commands:

- `f/l` - OPB input constraint loading
- `p/pol` - Justification via reverse Polish notation (RPN) arithmetic/cutting planes reasoning
- `u/rup` - Justification via reverse unit propagation (RUP)
- `o/soli` - Optimal value witness

In addition, we also support auxiliary VeriPB commands such as `a` and `j` (variants of implication that show up in proofs generated by the Glasgow solvers).

Our procedure performs a backwards reachability analysis similar to DRAT-trim [1]: first, we identify a minimal set of constraints required to justify the empty constraint $\bot = 0 \geq 1$, expressing unsatisfiability of the formula. We continue with our reachability analysis from this minimal set. For each lemma in the minimal set, we identify the lemmas needed to prove it, mark them as necessary for the proof, and add them to the set. All unmarked lemmas are discarded (trimmed).

The hinting/trimming algorithms can be partitioned into two sub-procedures: (1) RUP lemma hinting and (2) arithmetic/cutting planes reasoning.

1) Lemma Justification via RUP: The RUP procedure serves to simultaneously

- trim the formula by computing a minimal *necessary* set of constraints S' required to justify all lemmas in S.
- construct a set of hints mapping each constraint $c_i \in S'$ to a list of hints H_i, where each list element h_j is of the form $[d_j \ m_j]$, indicating that constraint d_j propagates unit m_j to falsify $\overline{c}_i$.

We make two notable optimizations here:

1) We implement a saturation procedure that aims to minimize the necessary set at each step—that is, we only consider new lemmas if we cannot justify our target with the current set.
2) Our data structures support near-instant unit discovery by utilizing properties of a constraint's slack: by maintaining a sorted order over our constraint database (terms sorted in decreasing-coefficient order, constraints sorted in decreasing-slack order), unit discovery amounts to queries to the first element under this ordering, which can be done efficiently.

2) Arithmetic/Cutting Planes Reasoning: The goal of the arithmetic procedures is to unroll sequences of reverse Polish notation (RPN) arithmetic into hinted chains of clausal reasoning. Notable optimizations include:

- General trie-based arithmetic simplification: we avoid recomputation of cutting-planes sequences that share common prefixes by maintaining a trie.
- Heuristic arithmetic trimming: the solvers we have tested tend to build up chains of cutting-planes reasoning where only the last element of the chain is useful for the final result, and hence the rest of the chain is unnecessary. We proceed with this assumption, but in event of propagation failure, we revert to a more cautious step-by-step processing.

B. BDD-Based PBIP Checking and LRAT Generation

Our goal is to create a TBDD representation $\dot{\mathbf{u}}_i$ for each constraint c_i in the proof sequence. Our implementation augments the existing TBDD operations in the TBUDDY package [20] to support PB constraints and to provide special operations to support RUP proof justification. The final step of adding infeasible constraint c_t will cause the empty clause to be added to the proof. Here we provide a high-level description of how the different PBIP steps translate into TBDD operations.

When adding constraint c_i, we invoke operation $\mathsf{BDD}(c_i)$ to construct the BDD representation $\mathbf{u}_i$ of c_i according to the algorithm described by Abío, et al. [32]. Upgrading this to the trusted BDD $\dot{\mathbf{u}}_i$ requires generating the unit clause $[u_i]$. We assume that every prior proof constraint $c_{i'}$, with $i' < i$, has a TBDD representation $\dot{\mathbf{u}}_{i'}$ with an associated unit clause $[u_{i'}]$.

When c_i is added by implication mode, generating its unit clause is based on the constraints given as the hint. If the hint consists of the single constraint $c_{i'}$, we can use the BDD_IMPLY operation to add proof clause $[\overline{u}_{i'} \vee u_i]$. Resolving this with the unit clause $[u_{i'}]$ then gives the unit clause $[u_i]$. When the hint consists of two constraints $c_{i'}$ and $c_{i''}$, we first use the BDD_AND operation on BDDs $\mathbf{u}_{i'}$ and $\mathbf{u}_{i''}$ to generate their conjunction $\mathbf{w}$, along with proof clause $[\overline{u}_{i'} \vee \overline{u}_{i''} \vee w]$. We then use the BDD_IMPLY operation to generate the clause $[\overline{w} \vee u_i]$. Resolving these clauses with the unit clauses for TBDDs $\dot{\mathbf{u}}_{i'}$ and $\dot{\mathbf{u}}_{i''}$ yields the unit clause $[u_i]$.

Adding constraint c_i via a RUP addition involves two phases. The first performs a series of clause generations to justify the unit propagations. The second uses a single clausal RUP addition to add the target clause. During the first phase, each step $j < k$ in the sequence $[d_1, m_1], [d_2, m_2], \ldots, [d_{k-1}, m_{k-1}], [d_k]$, requires generating a clause of the form $[u_i \vee \overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{j-1} \vee m_j]$. Step k requires generating the clause $[u_i \vee \overline{m}_1 \vee \overline{m}_2 \vee \cdots \vee \overline{m}_{k-1}]$. Implementing these justifications is complicated by the negations in the RUP steps, since negation is not directly supported in clausal reasoning. Instead, we make extensive use of DeMorgan's Laws. The final clausal RUP addition has unit clause $[u_i]$ as its target and will have as hints the unit-propagating clauses generated for the RUP steps. RUP addition will start with unit literal $\overline{u}_i$ and accumulate the propagated literals $m_1, m_2, \ldots, m_{k-1}$. The final clause will cause a conflict. For the special cases where either the previous constraint $c_{i'}$ or the target constraint c_i can be represented as a single clause, we can use this clause directly to justify unit propagation, reducing the number of BDD operations.

VI. Implementation and Results

The overall toolchain, illustrated in Figure 2, consists of the following steps

- IPBIP-HINTS: Translates from VeriPB to PBIP while simultaneously trimming the VeriPB proof, as described in Section V-A.
- PB-CNF: Generates a CNF representation of the input constraints by first constructing their BDD representations [32], and then encoding these with clauses, using at most two clauses per BDD node.
- PBIP-CHECK: Generates an LRAT file from the PBIP proof as described in Section V-B
- LRAT-CHECK: Checks an LRAT proof

As the thick lines in the figure indicate, steps PB-CNF and PBIP-CHECK can cause an exponential growth in the proof size, when input or intermediate constraints have large

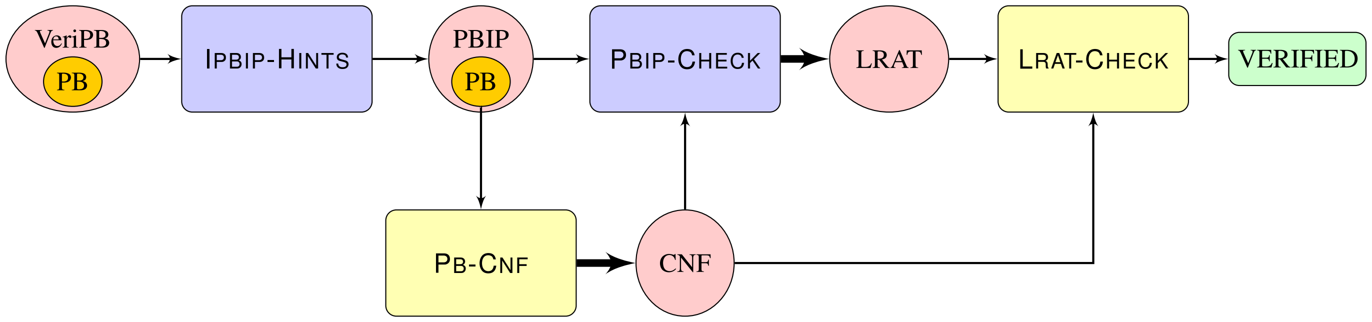


Fig. 2. PBIP toolchain: A VeriPB proof is first trimmed and translated into PBIP by IPBIP-HINTS. The VeriPB and the PBIP proofs contain a copy of the original PB problem, denoted by the orange PB subnodes. PB-CNF generates a CNF file from the original PB problem. The PBIP proof and CNF file are inputs to PBIP-CHECK, which translates the PBIP proof to LRAT using a a proof-generating BDD package [20]. The generated LRAT file can then be checked by LRAT-CHECK. The two thick lines indicate cases where there can be an exponential size increase. The yellow blocks indicate steps that must be correct to ensure soundness of the toolchain.

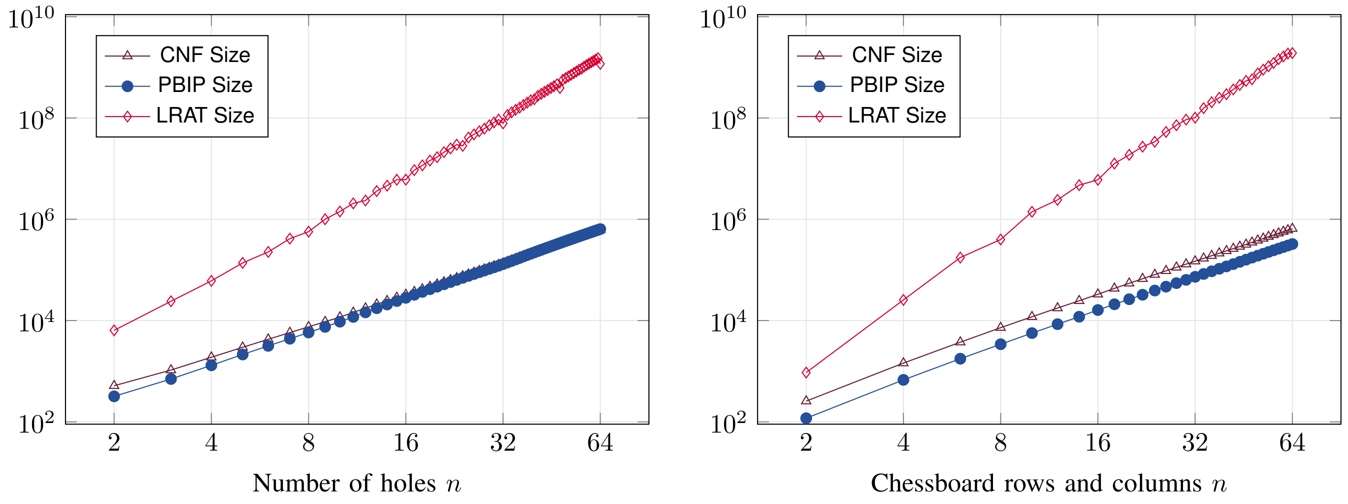


Fig. 3. File sizes in bytes for (left) pigeonhole and (right) mutilated chessboard

coefficients. The figure also indicates that steps PB-CNF and LRAT-CHECK form the trusted code base for the toolchain—they must be correct for the overall verification to be sound. In the case of LRAT-CHECK, one option would be to use a formally verified checker. In the case of PB-CNF, the tool is very simple, but it would be good to have a formally verified version, as is discussed in Section VII.

A. Benchmarks

We demonstrate the effectiveness of our tools (trimming procedure, clausal translation) and analyze our contributions by evaluating them on the following benchmark problems:

1) Pigeonhole (PHP) Formulas - PBIP proofs generated by summing the constraints across all pigeons and holes.
2) Mutilated Chessboard (MCB) Formulas - PBIP proofs generated by summing the constraints for every square.
3) DIMACS Clique (CLQ) Benchmarks (23 instances) - OPB/VeriPB proofs generated by Glasgow Clique Solver
4) Subgraph Isomorphism (SIP) Benchmarks (30 instances) - OPB/VeriPB proofs generated by Glasgow Subgraph Solver

For the CLQ benchmarks, the CakePB checker [22] was evaluated on a 55-benchmark subset of the Second DIMACS Implementation Challenge [36], out of which it was able to verify 50 graphs. From this subset of 55 benchmarks, we were able to translate 23 instances to LRAT and fully verify 21 of them with LRAT-CHECK.

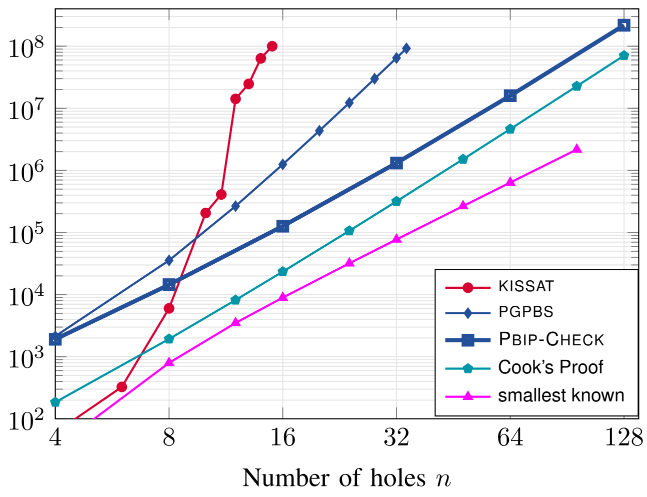
For the SIP benchmarks, we selected 30 instances from subgraph isomorphism problems hosted by Christien Solnon [37] and translate all of them down to LRAT and verify them with LRAT-CHECK.

We ran our tests on the Jetstream2 cluster hosting a system with a 16-core AMD EPYC-Milan Processor, 60GB RAM, and 1TB disk space running Ubuntu 22.04.3 LTS.

Our results can be broken down into three sets of experiments: (1) a comparison between PBIP/LRAT proof sizes and how our pipeline performs against competing tools, (2) an evaluation of the effectiveness of our trimming procedure, and (3) an analysis of the runtimes of our toolchain.

B. Proof Sizes

Pigeonhole (PHP)/Mutilated Chessboard (MCB) Benchmarks: For the pigeonhole and mutilated chessboard prob-



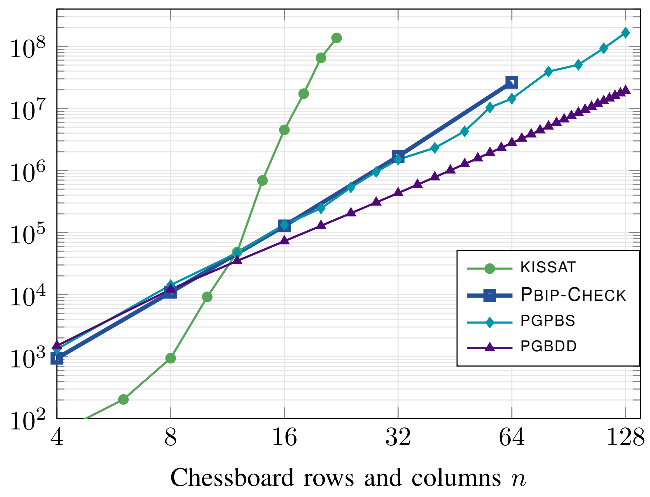


Fig. 4. Total number of clauses in proofs for (left) pigeonhole and (right) mutilated chessboard

lems, Figure 3 shows the different file sizes (in bytes) as a function of problem parameter n (the number of holes in PHP and the number of rows and columns in MCB). The graphs show a close correspondence between the CNF and the PBIP proof sizes, and a polynomial separation between the PBIP and LRAT proof sizes.

Figure 4 shows the number of clauses in proofs generated by our toolchain compared to those for proofs generated by competing tools. We consider here the proof-generating SAT solver KISSAT [38], the proof-generating pseudo-Boolean solver PGPBS [29], S. A. Cook's manually constructed $O(n^4)$ extended resolution proofs [39], and the smallest known proof ($O(n^3)$) [40]. We see that our PHP proofs asymptotically match the $O(n^4)$ scaling of Cook's proof. The scaling of the MCB proofs matches that of PGPBS but is bested by running the proof-generating BDD package PGBDD with a carefully devised variable ordering and sequencing of BDD operations [41]. In both instances, we greatly improve on the exponential performance of KISSAT.

Maximum Clique/Subgraph Isomorphism Benchmarks: We evaluate our pipeline on native pseudo-Boolean benchmarks (CLQ/SIP) starting from VeriPB cutting planes proofs. From the VeriPB proofs, we run the full pipeline presented in Figure 2 and obtain PBIP and LRAT proofs. Figure 5 compares the various proof sizes.

The general trend confirms the polynomial separation between the PBIP and LRAT proof sizes, with some irregularities due to extreme cases of proof trimming.

Figure 6 summarizes the average proof size increase (per benchmark suite) across the various proof formats in relation to the original VeriPB proof (in its non-kernel format).

C. Trimming Effectiveness

Here, we outline the effectiveness of the VeriPB trimming procedures described in Section V-A and implemented in IPBIP-HINTS.

Clique Benchmarks: Figure 7 demonstrates the effectiveness of our trimming on a set of DIMACS clique problems. On average, over our test suite, 45% of the input VeriPB lemmas are deemed unnecessary and are therefore trimmed from the proof. On 3 examples (`hamming8-2`, `c-fat500-10`, and `hamming10-2`), our trimmed (clausal) PBIP proofs are in fact shorter than the corresponding (non-clausal) VeriPB proofs. Notably, these cases are trimmed very aggressively (averaging 97.5% of lemmas trimmed) and this can be seen in Figure 5, where the VeriPB line rises above the PBIP line. The resulting LRAT proof for `hamming10-2` is also only 3.7 times larger than the corresponding VeriPB proof.

Subgraph Isomorphism Benchmarks: The proofs generated by the Glasgow Subgraph Solver [35] for the (unsatisfiable) subgraph isomorphism problems were succinct—most required only a single RUP justification amounting to the final unsatisfiability result. However, four benchmarks required more than one lemma. Notably, `g3-g12` underwent 91% trimming, with the corresponding LRAT file being comparable in size with the corresponding VeriPB file (as depicted in Figure 5, where the file sizes only differ by a factor of $1.3\times$).

D. Tool Runtimes

Clique Benchmarks: Our approach incurs a significant cost (in comparison with CakePB) in both the source trimming and the clausal translation, as shown in Figure 8. On easy instances, the trimming (IPBIP-HINTS, in blue) and checking (PBIP-CHECK in red and LRAT-CHECK in yellow) all perform moderately well whereas on hard instances, they become quite slow. Average ratios (in relation to CakePB's performance) over the benchmark sets are seen in Figure 9.

Subgraph Isomorphism Benchmarks: Similar to the clique benchmarks (as seen in Figure 8), our toolchain on subgraph isomorphism problems generally incurs a large runtime overhead versus CakePB. However, the trimming procedure does take less time than CakePB, amounting to 80% of CakePB's total runtime. This can be attributed to the succinctness of the

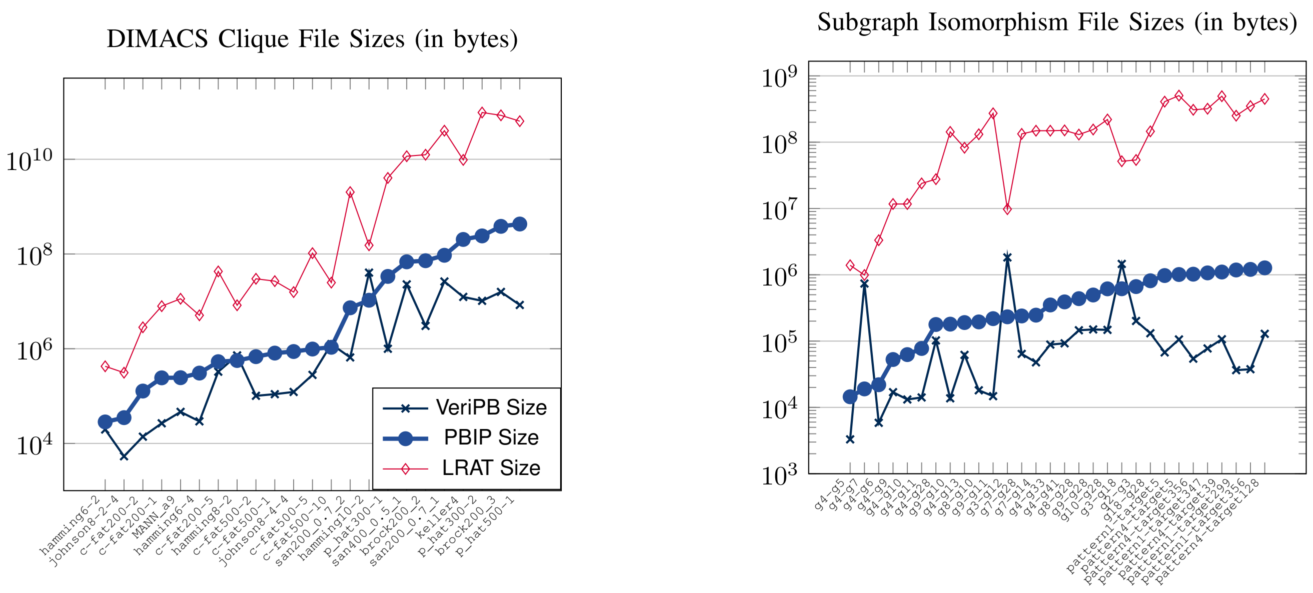


Fig. 5. (L) DIMACS MAX-Clique proof file sizes in bytes (R) Subgraph Isomorphism proof file sizes in bytes

Benchmark	VeriPB Kernel	PBIP	LRAT
CLQ	1.9×	11.3×	1682.0×
SIP	2.8×	8.1×	3342.8×

Fig. 6. Average proof size increase between VeriPB (non-kernel) and Kernel/PBIP/LRAT

generated VeriPB proofs by the subgraph solver, requiring less effort from our trimming/propagation procedures.

VII. Conclusion and Future Work

We have presented a pipeline capable of translating native pseudo-Boolean proofs (in the VeriPB format) to extended resolution proofs (in the LRAT format). This involved introducing the intermediate PBIP (Pseudo-Boolean Implication Proof) framework, from which a proof-generating BDD package can generate the LRAT proof.

The work reported here suggests several avenues for future research.

Verified PB Encodings. As indicated in Figure 2, we use the unverified program PB-CNF to generate a CNF representation of the input constraints. Although generating a clausal representation of pseudo-Boolean constraints is straightforward, this still represents a weak link in terms of the trustworthiness of our toolchain. Based on recent work on formalized CNF encodings in the Lean proof framework [42], we could produce verified CNF encodings of PB constraints expressed in the OPB format and achieve end-to-end verification (verified encodings of the PB source, verified translation by PBIP-CHECK, and verified checking via LRAT-CHECK) of our pipeline.

Supporting a Larger Subset of VeriPB. VeriPB is capable of even richer modes of reasoning, supporting rules such as redundancy-based strengthening and dominance-based strengthening [21]. Converting these to BDD-based proofs

benchmark	total u	done u	trimmed (%)
brock200_2	3758	3388	9.85
brock200_3	14251	14210	0.29
c-fat200-1	17	6	64.71
c-fat200-2	3	1	66.67
c-fat200-5	86	29	66.28
c-fat500-1	9	2	77.78
c-fat500-2	15	2	86.67
c-fat500-5	34	2	94.12
c-fat500-10	65	1	98.46
hamming6-2	17	1	94.12
hamming6-4	82	82	0.0
hamming8-2	65	2	96.92
hamming10-2	257	3	98.83
johnson8-2-4	24	24	0.0
johnson8-4-4	120	115	4.17
keller4	13542	13495	0.35
MANN_a9	71	53	25.35
p_hat300-1	1473	1434	2.65
p_hat300-2	4078	3367	17.44
p_hat500-1	9708	9677	0.32
san200_0.7_1	13396	2604	80.56
san200_0.7_2	450	246	45.33
san400_0.5_1	2276	1554	31.72
g2-g3	701	701	0.0
g3-g12	411	37	91.0
g3-g18	321	69	78.5
g4-g7	21	20	4.76

Fig. 7. VeriPB lemmas trimmed on CLQ/SIP benchmarks - The column marked "total u" represents the number of lemmas present in the source VeriPB proof (RUP lemmas logged by the Glasgow solvers in their derivations of $\perp$) and "done u" represents the number of RUP lemmas *actually* deemed necessary by our IPBIP-HINTS trimming procedure. The top 23 benchmarks are max-clique benchmarks while the bottom 4 are (selected) subgraph isomorphism benchmarks.

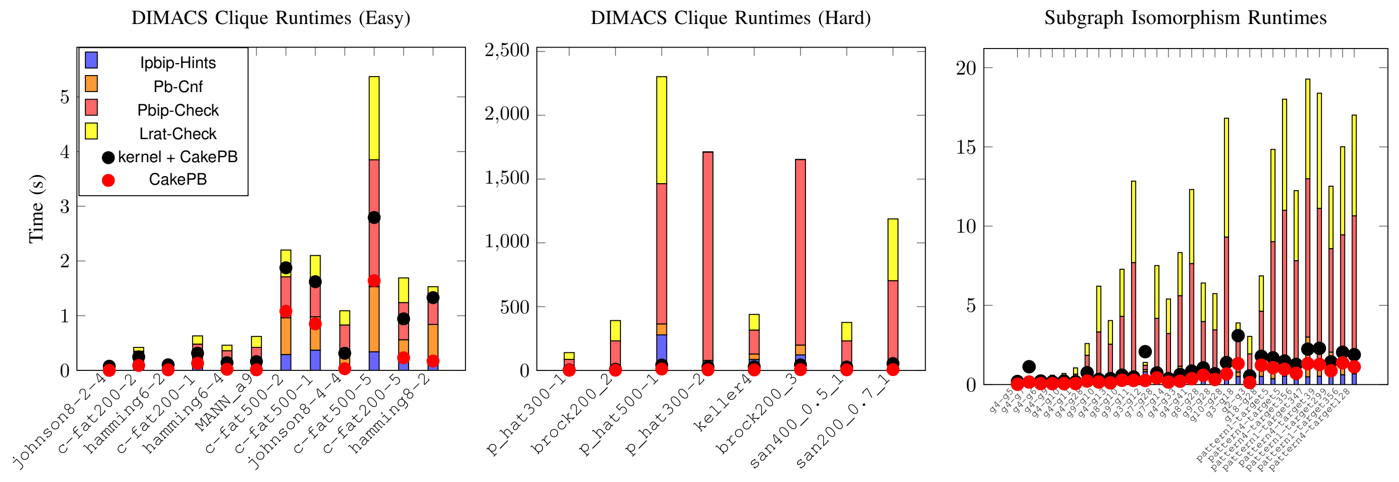


Fig. 8. (L) Toolchain performance on (easy) DIMACS Max-Clique benchmarks (M) Toolchain Performance on (hard) DIMACS Max-Clique benchmarks (R) Toolchain performance on (selected) Subgraph Isomorphism benchmarks
The red dot (labelled CakePB) corresponds to running solely the CakePB checker on an already-generated kernel format while the black dot (labelled kernel + CakePB) incorporates the time taken to generate the kernel format as well.
Note: `lrat_check` was unable to verify `p_hat300-2` and `brock200_3` from the middle (hard cliques) graph.

Set	Ipbip-Hints	Pbip-Check	Lrat-Check
sip	0.8×	9.5×	6.9×
clq (all)	4.4×	42.5×	10.7×
clq (easy)	0.9×	4.2×	2.6×
clq (hard)	10.9×	109.5×	35.3×

Fig. 9. Average runtime overhead (ratio) of each phase in comparison with CakePB checking.

would require going beyond the implication-based proofs supported by current proof-generating BDD packages. It requires having proof rules that allow adding clauses that preserve satisfiability but exclude possible solutions to a formula, such as propagation redundancy [5], [43]. Translating the PB strengthening rules into clausal proofs remains an unsolved problem.

Fine tuning Performance/Tool Heuristics. Our tools make use of various heuristics (from proof-specific optimizations for trimming to BDD variable orderings). Fine-tuning these and optimizing relevant parts of the toolchain (more efficient structures for trimming, cache optimization) is definitely of interest and could see improvements in the tool runtimes described in Section VI-D.

Improvements to CakePB. Several of the optimizations we made in our toolchain could be applied to CakePB:

- *Direct support for RUP*. The CakePB toolchain requires converting a VeriPB proof into kernel format, replacing each RUP addition with a sequence of cutting-planes operations. Our experimental results show that this generally causes a small expansion in the proof size and a small time overhead, but it is awkward, and it prompted the authors to introduce special provisions to help users debug failed proofs [22]. We have shown that methods similar to those used by DRAT-trim [1] can be used to identify the unit propagation steps required to justify a RUP addition. These steps could be checked directly by CakePB.
- *Proof trimming*. Our experimental results show that many of the steps in VeriPB proofs are not relevant for a proof of unsatisfiability. Trimming these can reduce the checking time. It can also enable generating an "unsat core" identifying the key properties of the problem that cause it to be unsatisfiable. This capability has many applications beyond proof generation [44].

Acknowledgments

The authors thank the MIAO group and their collaborators—in particular Ciaran McCreesh, Andy Oertel, and Yong Kiam Tan–for support with their tools, benchmarks, and general advice regarding pseudo-Boolean solving. Special thanks to Ciaran McCreesh for numerous in-depth clarifications on these matters. In addition, the authors acknowledge Ruben Martins and Joseph Reeves of Carnegie Mellon for helpful advice and encouraging discussions over the course of the project. Finally, the authors thank Stephen Deems of the Pittsburgh Supercomputing Center for providing the computing resources to run our experiments.

This work was supported by the U. S. National Science Foundation under grant CCF-2108521.

References

[1] M. J. H. Heule, "The DRAT format and DRAT-trim checker," *arXiv preprint arXiv:1610.06229*, 2016.

[2] A. Balint, M. J. H. Heule, A. Belov, and M. Järvisalo, "The application and the hard combinatorial benchmarks in SAT competition 2013," *Proceedings of SAT Competition*, pp. 99–100, 2013.

[3] M. J. H. Heule, W. Hunt, M. Kaufmann, and N. Wetzler, "Efficient, verified checking of propositional proofs," in *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8*, pp. 269–284, Springer, 2017.

[4] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pp. 220–236, Springer, 2017.

[5] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "Verified propagation redundancy and compositional UNSAT checking in CakeML," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 2, pp. 167–184, 2023.

[6] M. J. H. Heule, "Schur number five," *CoRR*, vol. abs/1711.08076, 2017.

[7] P. M. Dearing, P. L. Hammer, and B. Simeone, "Boolean and graph theoretic formulations of the simple plant location problem," *Transportation Science*, vol. 26, no. 2, pp. 138–148, 1992.

[8] Y. Crama and P. L. Hammer, "Recognition of quadratic graphs and adjoints of bidirected graphs," in *Proceedings of the third international conference on Combinatorial mathematics*, pp. 140–149, 1989.

[9] P. L. Hammer and E. Shlifer, "Applications of pseudo-Boolean methods to economic problems," *Theory and decision*, vol. 1, pp. 296–308, 1971.

[10] F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt, "An application of combinatorial optimization to statistical physics and circuit layout design," *Operations Research*, vol. 36, no. 3, pp. 493–513, 1988.

[11] F. A. Aloul, A. Ramani, I. Markov, and K. Sakallah, "PBS: a backtrack-search pseudo-Boolean solver and optimizer," in *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pp. 346–353, 2002.

[12] D. Chai and A. Kuehlmann, "A fast pseudo-Boolean constraint solver," in *Proceedings of the 40th annual Design Automation Conference*, pp. 830–835, 2003.

[13] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-Boolean SAT solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 1-4, pp. 165–189, 2006.

[14] J. Elffers and J. Nordström, "Divide and conquer: Towards faster pseudo-Boolean solving.," in *IJCAI*, vol. 18, pp. 1291–1299, 2018.

[15] S. Gocht, R. McBride, C. McCreesh, J. Nordström, P. Prosser, and J. Trimble, "Certifying solvers for clique and maximum common (connected) subgraph problems," in *Principles and Practice of Constraint Programming (CP)*, 2020.

[16] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint program solver," in *Principles and Practice of Constraint Programming (CP)*, 2022.

[17] S. Gocht, *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, 2022.

[18] G. S. Tseitin, "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.

[19] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[20] R. E. Bryant, "TBUDDY: A proof-generating BDD package," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 49–58, IEEE, 2022.

[21] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified dominance and symmetry breaking for combinatorial optimisation," *Journal of Artificial Intelligence Research*, 2023.

[22] S. Gocht, C. McCreesh, M. O. Myreen, J. Nordström, A. Oertel, and Y. K. Tan, "End-to-end verification for subgraph solving," in *AAAI Conference on Artificial Intelligence*, 2024.

[23] B. Bogaerts, C. McCreesh, M. O. Myreen, J. Nordström, A. Oertel, and Y. K. Tan, "Documentation of VeriPB and CakePB for the SAT competition 2023 (Mar 2023)." https://satcompetition.github.io/2023/downloads/proposals/veripb.pdf.

[24] W. J. Cook, C. R. Coullard, and G. X. R. Turán, "On the complexity of cutting-plane proofs," *Discrete Applied Mathematics*, vol. 18, pp. 25–38, 1987.

[25] S. A. Cook, "Feasibly constructive proofs and the propositional calculus," in *ACM Symposium on the Theory of Computing (STOC)*, pp. 83–97, 1975.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W. H. Freeman and Company, 1979.

[27] R. E. Bryant and M. J. H. Heule, "Generating extended resolution proofs with a BDD-based SAT solver," *ACM Transactions on Computational Logic*, vol. 24, no. 4, pp. 1–28, 2023.

[28] R. E. Bryant and M. J. H. Heule, "Dual proof generation for quantified Boolean formulas with a BDD-based solver," in *Conference on Automated Deduction (CADE)*, vol. 12699 of *LNAI*, pp. 433–449, 2021.

[29] R. E. Bryant, A. Biere, and M. J. H. Heule, "Clausal proofs for pseudo-Boolean reasoning," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 443–461, Springer, 2022.

[30] C. Sinz and A. Biere, "Extended resolution proofs for conjoining BDDs," in *Computer Science Symposium in Russia (CSR)*, vol. 3967 of *LNCS*, pp. 600–611, 2006.

[31] T. Jussila, C. Sinz, and A. Biere, "Extended resolution proofs for symbolic SAT solving with quantification," in *Theory and Applications of Satisfiability Testing (SAT)*, vol. 4121 of *LNCS*, pp. 54–60, 2006.

[32] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "A new look at BDDs for pseudo-Boolean constraints," *Journal of Artificial Intelligence Research*, vol. 45, pp. 443–480, 2012.

[33] O. Roussel and V. Manquinho, "Input/output format and solver requirements for the competitions of pseudo-Boolean solvers." https://www.cril.univ-artois.fr/PB12/format.pdf, 2012.

[34] J. Elffers and J. Nordström, "Divide and conquer: Towards faster pseudo-boolean solving.," in *IJCAI*, vol. 18, pp. 1291–1299, 2018.

[35] C. McCreesh, P. Prosser, and J. Trimble, "The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants," in *International Conference on Graph Transformation*, pp. 316–324, Springer, 2020.

[36] D. S. Johnson and M. A. Trick, *Cliques, coloring, and satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993*, vol. 26. American Mathematical Soc., 1996.

[37] C. Solnon, "Benchmarks for the subgraph isomorphism problem." http://liris.cnrs.fr/csolnon/SIP.html, 2016. visited on May 11th, 2024.

[38] A. Biere and M. Fleury, "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022," in *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions* (T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2022-1 of *Department of Computer Science Series of Publications B*, pp. 10–11, University of Helsinki, 2022.

[39] S. A. Cook, "A short proof of the pigeon hole principle using extended resolution," *Acm Sigact News*, vol. 8, no. 4, pp. 28–32, 1976.

[40] I. Grosof, N. Zhang, and M. J. H. Heule, "Towards the shortest DRAT proof of the pigeonhole principle," 2022.

[41] R. E. Bryant and M. J. H. Heule, "Generating extended resolution proofs with a BDD-based SAT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I*, vol. 12651 of *LNCS*, pp. 76–93, 2021.

[42] C. R. Codel, J. Avigad, and M. J. H. Heule, "Verified encodings for SAT solvers," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 141–151, IEEE, 2023.

[43] M. J. H. Heule, B. Kiesl, and A. Biere, "Strong extension-free proof systems," *Journal of Automated Reasoning*, 2019.

[44] J. P. Marques-Silva, "Minimal unsatisfiability: Models, algorithms, and applications," in *IEEE Symposium on Multi-Valued Logic*, 2010.

Formal Methods in Computer-Aided Design 2024

Verified Substitution Redundancy Checking

Cayden R. Codel
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
ccodel@cs.cmu.edu

Jeremy Avigad
Department of Philosophy
Carnegie Mellon University
Pittsburgh, USA
avigad@cmu.edu

Marijn J. H. Heule
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
marijn@cmu.edu
Amazon Scholar

***Abstract*—Modern SAT solvers are trustworthy because their results can be expressed in formal proof systems and validated with verified proof checkers. Today, the RAT and PR proof systems are the de facto standard: they capture many reasoning techniques used by SAT solvers, and they are supported by efficient, formally-verified checkers. However, RAT and PR struggle to succinctly express advanced reasoning techniques like symmetry breaking.**

In this paper, we present proof checking tools for the substitution redundancy (SR) proof system, a powerful generalization of PR and RAT. We first highlight three problems with linear-size SR proofs that are not expected to have linear-size PR or RAT proofs. We then present proof formats for SR with and without unit propagation hints, a tool to add those hints, and the first verified SR proof checker. Since SR generalizes other proof systems, our checker has the distinction of supporting the strongest clausal proof system to date. Finally, our experimental results show that SR proofs are much smaller than their RAT counterparts, and that verified SR proof checking is efficient in practice.

I. Introduction

Satisfiability (SAT) solving continues to be a crucial tool for industry and academia. For example, SAT solvers were recently used to resolve open problems in computational geometry [1, 2] and to improve lower bounds for a problem in quantum mechanics [3]. In addition, SAT solvers form the core of SMT solvers, which are queried a billion times a day by various Amazon Web Services applications [4].

SAT solving is *trustworthy* due to the development of verified proof checking. When reporting that a problem has no solutions, modern SAT solvers emit a corresponding *proof of unsatisfiability* expressed in a formal *proof system*. By validating these proofs with verified software, we gain a high degree of confidence in solver results, particularly those backing mathematical theorems and industrial security guarantees.

Proof systems and SAT solvers are complementary, with advances in one driving innovations in the other. For example, the RAT proof system [5] was developed to validate the learned clauses produced by CDCL solvers and some inprocessing techniques. In the other direction, the PR proof system [6], a generalization of RAT, validates short clauses that solvers could not initially derive. But solvers have since caught up: the 2nd- and 8th-place finishers at the 2023 SAT Competition used PR reasoning as a preprocessing step [7, 8]. More generally, stronger proof systems enable solvers to use more-powerful reasoning techniques and produce shorter proofs.

In this paper, we develop verified proof checking tools for the substitution redundancy (SR) proof system [9–11] (Section II). SR generalizes PR, such that it can succinctly express a broad range of symmetry-breaking techniques that PR cannot. We show this by highlighting three problems that have SR proofs with size linear in the number of variables and that, as far as we know, do not have linear-sized PR proofs (Section III).

Currently, no solver supports SR reasoning. However, we expect that the availability of our fast, verified SR checker will stimulate the development of SR reasoning and preprocessing techniques, similar to what happened with PR.

To enable SR proof checking (Section IV), we introduce the DSR and LSR proof formats (Section V). Like the formats for RAT and PR, DSR proofs record basic proof steps, while LSR proofs include unit propagation hints that guide proof checking. These formats are backwards compatible with the ones for RAT and PR. Our set of SR proof checking tools (Section VI) includes a tool that converts DSR proofs into DRAT and a tool that converts DSR proofs into LSR by adding hints.

We also present the first verified LSR proof checker (Section VII), giving it the distinction of supporting the strongest clausal proof system to date. It implements several data structures and techniques commonly used in SAT proof checkers, and we discuss how they impacted our formalization. We proved our checker correct with the Lean theorem prover [12].

Finally, our experimental results (Section VIII) show the clear benefits of using a stronger proof system. For example, we found that the SR proofs from our benchmarks had 0.4% as many proof lines as their transformations into RAT. We also show that verified SR proof checking is efficient in practice: our verified checker performs similarly to cake_lpr [13], a fast, verified PR proof checker written in CakeML [14].

II. Substitution Redundancy

We assume that the reader is familiar with SAT.[1] Throughout, let F be a formula in conjunctive normal form (CNF), let C and D be clauses of boolean literals, and let τ be a truth assignment on the boolean variables in F. We write $\overline{\ell}$ for the negation of boolean literal ℓ, and $\neg C := \bigwedge_{\ell \in C} \overline{\ell}$ for the negation of a clause C. A *partial truth assignment* is a non-tautological

Many thanks to the researchers on the Lean theorem prover Zulip chatroom. Our research was supported by NSF grant CCF-2229099, AFRL agreement FA8750-24-9-1000, and the Hoskinson Center for Formal Mathematics.

[1] For background reading, see Ch. 15 of the Handbook of Satisfiability [15].

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_24

set of literals taken to be true. We abuse notation by writing evaluation under (partial) assignments as $\tau(x)$.

When a SAT solver claims that a formula F is unsatisfiable, it emits a proof of unsatisfiability in a formal proof system. In *clausal proof systems*, each proof step adds a clause C to F such that F and $F \land C$ are *equisatisfiable*, meaning that F is satisfiable if and only if $F \land C$ is. Such clauses are called *redundant*. The backward direction is trivial, so it suffices to show only the forward direction. Notably, adding C to F may (and often does) reduce the set of satisfying assignments, but the important thing is that satisfiability is preserved. By producing a series of redundant clauses ending in the empty clause $\bot$, a SAT solver can show that F is unsatisfiable.

Unfortunately, it is NP-hard to determine whether an arbitrary C is redundant for F [16]. Therefore, most clausal proof systems instead use a property implying redundancy that is checkable in polynomial time when provided with a *witness*. The SR proof system is based on such a property.

Witnesses for clausal proof systems work as follows. Suppose we are trying to show that C is redundant for F. If F *entails* C, written as $F \vDash C$ and meaning that for any $\tau \vDash F$, we have that $\tau \vDash C$, then certainly F and $F \land C$ are equisatisfiable. So assume that τ satisfies F but does not satisfy C. We can show that C is redundant by modifying τ into a new assignment satisfying $F \land C$. The witness σ describes this modification, and it is used to form the satisfying truth assignment $\tau \circ \sigma$.

As a motivating example (discussed further in Section III), consider the following proof of unsatisfiability for the pigeonhole formula on n pigeons and $n-1$ holes. To start, we learn that the first pigeon p_1 cannot go in the last hole h_{n-1}, i.e., the unit clause $\overline{x}_{1,n-1}$. If $F \vDash \overline{x}_{1,n-1}$, then we'd be done, so assume that $\tau \vDash F$ but $\tau \vDash x_{1,n-1}$, meaning that τ places p_1 in h_{n-1}. One way of modifying τ to satisfy $\overline{x}_{1,n-1}$ is to swap p_1 with a different pigeon, say, p_n, thus ensuring that p_1 ends up in a different hole while still satisfying the overall formula. We accomplish this by mapping the variables associated with p_1 to the variables associated with p_n, and vice versa, before evaluating them under τ. Functions called substitutions capture this technique, and they serve as the witnesses in SR.

Formally, a *substitution* σ maps boolean variables to either a boolean literal or a truth value. They can satisfy clauses and formulas, written as $\sigma \vDash C$ and $\sigma \vDash F$. They can also reduce them. The *reduction* of C under σ, written as $C_{|\sigma}$, is obtained by mapping σ over its literals and removing those falsified by σ. The reduction of F under σ, written as $F_{|\sigma}$, is obtained by reducing each of its clauses and removing those satisfied by σ. We say that σ *reduces* C if σ does not satisfy C and there is a literal $\ell \in C$ not mapped to itself under σ, i.e., $\sigma(\ell) \neq \ell$. Notably, σ can reduce C even if $C_{|\sigma} = C$, as in the example where $C := x_1 \lor x_2$ and σ maps x_1 and x_2 to each other.

Additionally, we can compose substitutions with truth assignments to form new truth assignments. Define $(\tau \circ \sigma)(x)$ as $\sigma(x)$ if $\sigma(x) \in \{\top, \bot\}$ and as $\tau(\sigma(x))$ otherwise. From this definition, we can derive the core lemma used to prove redundancy from the SR property: if σ does not satisfy C, then $\tau \circ \sigma \vDash C \Leftrightarrow \tau \vDash C_{|\sigma}$. In other words, knowing that τ satisfies $C_{|\sigma}$ is the same as knowing that σ modifies τ into an assignment $(\tau \circ \sigma)$ satisfying C. The lemma follows from the definition of composition: let $\ell \in C$ with $\tau(\sigma(\ell)) = \top$. But since $\sigma \nvDash C$, then $\sigma(\ell) \in C_{|\sigma}$, and so $\tau \vDash C_{|\sigma}$.

The SR property is based on *unit propagation* (UP), which computes in polynomial time the partial assignment implied by the unit clauses in a formula F. Starting from the empty partial assignment τ, UP reduces F with the following rule until fixpoint: if $F_{|\tau}$ contains a unit clause x, then $\tau := \tau \cup x$. If UP finds a unit clause $x \in F_{|\tau}$ with $\tau(x) = \bot$, then we write $F \vdash_1 \bot$, and we say that we have a *UP refutation* for F. Such formulas are unsatisfiable.

We extend this definition to include UP derivations of clauses and formulas. If C is a clause, then $F \vdash_1 C$ if $F \land \neg C \vdash_1 \bot$. Likewise, if G is a formula, then $F \vdash_1 G$ if $F \vdash_1 C$ for every $C \in G$. It is well-known that if $F \vdash_1 G$, then $F \vDash G$.

We now define SR. A clause C is *substitution redundant* (SR) [9–11] for a formula F if there exists a substitution σ such that $F \land \neg C \vdash_1 (F \land C)_{|\sigma}$.[2]

The SR property implies redundancy.

Theorem 1 ([9, 10]). *Let F be a CNF formula, and let C be a clause. If C is SR for F, then C is redundant for F.*

Proof. It suffices to show the forward direction. If $F \vDash C$, then we'd be done, so assume that $\tau \vDash F$ and $\tau \vDash \neg C$, and let σ be a substitution satisfying the SR property for C and F. We will show that $\tau \circ \sigma \vDash F \land C$, meaning that $\tau \circ \sigma \vDash D$ for every $D \in (F \land C)$. If $\sigma \vDash D$, then so would $\tau \circ \sigma \vDash D$ and we'd be done, so assume otherwise. Thus $D_{|\sigma} \in (F \land C)_{|\sigma}$. But the SR property tells us that since $\tau \vDash F \land \neg C$, we have that $\tau \vDash D_{|\sigma}$, and by the lemma, this implies that $\tau \circ \sigma \vDash D$. ☐

SR generalizes PR [6], which itself generalizes RAT [5].[3] If we restrict the witness to be a partial truth assignment, we obtain PR. If we restrict the witness further to be a partial assignment defined by a single literal, we obtain RAT.

III. Short SR proofs

Many problems, including several that are hard for resolution, have SR proofs with size linear in the number of variables. In this section, we describe SR proofs for three such problems that, as far as we know, do not have linear-sized PR proofs. Our manually-constructed proofs are available at:

`https://github.com/marijnheule/sr-proofs.`

A. The pigeonhole principle

The pigeonhole principle (PHP) states that if n pigeons are placed in $m < n$ holes, then at least one hole contains multiple pigeons. The unsatisfiable PHP CNF formulas on n pigeons and $n-1$ holes consist of $O(n^2)$ variables $\{x_{p,h}\}$, where $x_{p,h} = \top$ means that pigeon p is in hole h, and $O(n^3)$ clauses,

[2] There are several variants of SR. We use the one due to Gocht and Nordström [10] because it is more general and easier to understand than the original definition due to Buss and Thapen [9], which requires that $\sigma \vDash C$ or that $C_{|\sigma}$ is a tautology, and that $F_{|\tau} \vdash_1 F_{|\sigma}$.

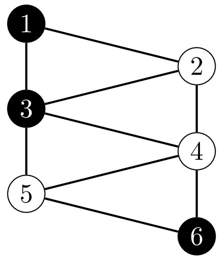
[3] These proof systems, and their extension-free variants, form a proof hierarchy with interesting proof-theoretic properties [9, 11].

encoding that each pigeon must be in at least one hole and that two pigeons cannot both be in the same hole. Resolution proofs of PHP formulas are exponential in n [17]. Extended resolution admits proofs of size $O(n^4)$ [18], while PR admits proofs of size $O(n^3)$ [16].

PHP SR proofs consist of $O(n^2)$ unit clauses [9, 11]. They recursively use the following scheme to show that pigeon p_n must go in hole h_{n-1}. We start by learning that p_1 does not go in the last hole, i.e., the unit clause $\overline{x}_{1,n-1}$. This clause is SR because if the satisfying assignment τ assigns p_1 to h_{n-1}, then we may modify τ with the substitution σ that instead assigns p_n to h_{n-1} by swapping (the variables for) the two pigeons. Formally, $\sigma = \{x_{1,n-1} \mapsto \bot,\ x_{n,n-1} \mapsto \top,\ x_{1,i} \mapsto x_{n,i},\ x_{n,i} \mapsto x_{1,i}\}$ for $i \in \{1, \ldots, n-2\}$. Next we learn that p_2 does not go in the last hole by swapping p_2 with p_n, and so on, until only p_n can go in h_{n-1}. Now the problem has been reduced to the PHP formula on $n-1$ pigeons. Repeating this process $n-1$ times results in a refutation.

B. Tseitin formulas for expander graphs

Given a simple, undirected graph with an odd number of black vertices and all others colored white, Tseitin formulas ask whether there exists a subset of the edges S such that every black vertex has odd degree in S and every white vertex has even degree in S. This is not possible by the handshake lemma: the sum of all vertex degrees in S is even, since each edge is counted twice, but the sum of black edges (odd) and white edges (even) must be odd. In the formula, every edge e receives a variable that encodes whether $e \in S$. Figure 1 illustrates the Tseitin constraints for a small graph. Tseitin formulas of expander graphs have exponentially-large resolution proofs [19] and polynomial-sized PR proofs [20].



$e_{1,2} \oplus e_{1,3} = 1$

$e_{1,2} \oplus e_{2,3} \oplus e_{2,4} = 0$

$e_{1,3} \oplus e_{2,3} \oplus e_{3,4} \oplus e_{3,5} = 1$

$e_{2,4} \oplus e_{3,4} \oplus e_{4,5} \oplus e_{4,6} = 0$

$e_{3,5} \oplus e_{4,5} \oplus e_{5,6} = 0$

$e_{4,6} \oplus e_{5,6} = 1$

Fig. 1. The Tseitin constraints for a small graph, ordered by vertex. If $e_{i,j} = \top$, then $e_{i,j}$ is in the edge subset S. The symbol $\oplus$ denotes XOR.

SR does better: any Tseitin formula has an SR proof linear in the number of edges using the following derivation. First, we may remove any vertex v incident to only a single edge e because if v is white, then e cannot be in S, so removing them both does not affect any other constraint; and if v is black, then e must be in S, so we may remove them both as long as we flip the color of the neighbor of v, since removing e will change the degree-parity of the neighbor. We keep removing degree-one vertices until a conflict (i.e., a black vertex with no edges) or until every vertex in the graph has degree at least two. Such a graph must have a cycle U. For any satisfying S, we may swap the membership of every edge $e \in U$ in S and get a new satisfying edge set, as doing so maintains the S-degree-parity of every vertex in U. As a result, we may pick an arbitrary edge $e \in U$ and fix $e \notin S$. Repeating this process will eventually result in a conflict.

We illustrate this process with the graph in Figure 1. Consider the cycle $\{e_{1,2},\ e_{1,3},\ e_{2,3}\}$. We can learn that the unit clause $\overline{e}_{1,2}$ is SR with the witness $\sigma = \{e_{1,2} \mapsto \bot,\ e_{1,3} \mapsto \overline{e}_{1,3},\ e_{2,3} \mapsto \overline{e}_{2,3}\}$. Afterwards, vertex v_1 is adjacent to only the edge $e_{1,3}$. Since v_1 is colored black, $e_{1,3}$ must be in our edge set S, so we may remove v_1 and $e_{1,3}$ to make the graph smaller, as long as we swap the color of v_3 from black to white.

C. Ramsey numbers

Ramsey number $R(k, \ell)$ is the smallest n such that every 2-coloring of the edges of the fully-connected graph on n vertices with the colors red and blue has either a red k-clique or a blue ℓ-clique. The encoding of $R(k, \ell)$ is straightforward: boolean variables $\{e_{i,j}\}_{1 \le i < j \le n}$ represent the color of each edge, where $e_{i,j} = \top$ means the edge is blue, and for each k-clique and ℓ-clique, there is a clause stating that at least one of its edges is blue or red, respectively. An unsatisfiable formula for any n, k, and ℓ proves that $R(k, \ell) \le n$.

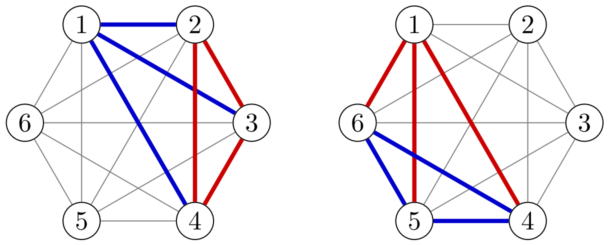


Fig. 2. A short proof of $R(3, 3) \le 6$. After adding clauses that sort the edges of vertex v_1 by color (blue edges first), fixing the edge $e_{1,4}$ to either blue or red forces a red or blue 3-clique, respectively, via unit propagation.

Short SR proofs for small Ramsey numbers can be constructed by sorting edges by color. For example, we can assume that the blue edges for vertex v_1 come first, represented by the clauses $e_{1,j} \vee \overline{e}_{1,j+1}$ for $1 < j < n$. Figure 2 illustrates the refutation derived by adding these four clauses to the formula for $R(3, 3) \le 6$. These binary clauses are SR. For instance, symmetry-breaking clause $e_{1,2} \vee \overline{e}_{1,3}$ has witness $\sigma = \{e_{1,2} \mapsto \top,\ e_{1,3} \mapsto \bot,\ e_{2,4} \mapsto e_{3,4},\ e_{3,4} \mapsto e_{2,4},\ e_{2,5} \mapsto e_{3,5},\ e_{3,5} \mapsto e_{2,5},\ e_{2,6} \mapsto e_{3,6},\ e_{3,6} \mapsto e_{2,6}\}$.

Ramsey number $R(4, 4) = 18$. A resolution proof of $R(4, 4) \le 18$ requires around a billion resolution steps. In contrast, the SR proof uses only 38 clause addition steps. See Section X for the argument.

IV. SR Proof Checking

In this section, we discuss the SR proof checking algorithm and why it checks redundancy, as well as two performance bottlenecks that are addressed in our verification.

The algorithm (Algorithm 1) is divided into two phases. The first phase (Lines 1–4) determines whether $F \vdash_1 C$, which would imply that $F \vDash C$, and thus that C is redundant. By definition, this amounts to showing that $F \wedge \neg C \vdash_1 \bot$. We start with the partial assignment $\tau := \neg C$ (Line 1) and then try to find a UP refutation for F (Lines 2–4).

Algorithm 1: Validating whether a clause is SR

```
   Input: CNF $F$, clause $C$, and witness $\sigma$ satisfying $C$
   Output: "Yes" if $C$ is SR for $F$, "No" otherwise.
1  Set $\tau \leftarrow \neg C$
2  while there is a $D \in F_{|\tau}$ with $|D| \leq 1$ do          \
3      if $|D| = 0$ (i.e., $D = \bot$) then return Yes               } UP
4      else  $\tau \leftarrow \tau \cup D$                          /
5  if $C = \bot$ then return No
6  for $D \in F$ do
7      if $\sigma \vDash D$ or $D$ is not reduced by $\sigma$ then continue
8      if $\tau \vDash D_{|\sigma}$ then continue
9      $\tau' \leftarrow \tau \cup \neg D_{|\sigma}$
10     while there is an $E \in F_{|\tau'}$ with $|E| \leq 1$ do    \
11         if $|E| = 0$ (i.e., $E = \bot$) then                     |
12             continue to the next iteration of Line 6              } UP
13         else  $\tau' \leftarrow \tau' \cup E$                    /
14     return No
15 return Yes
```

If no UP refutation is found, then τ stores the unit clauses found by UP on F, and we proceed to the second phase, the SR check (Lines 5–15). The empty clause $\bot$ cannot be SR, so we stop if $C = \bot$ (Line 5). Otherwise, we check the SR property. In our proof checking tools, we assume that the witness σ satisfies C, so it suffices to show that $F \wedge \neg C \vdash_1 D_{|\sigma}$ for each reduced clause $D_{|\sigma} \in F_{|\sigma}$.

The actual SR check looks slightly different. Rather than iterate across every $D_{|\sigma}$, we instead iterate across every $D \in F$ (Line 6), but we skip some clauses (Line 7). If $\sigma \vDash D$, then it is not in $F_{|\sigma}$; and if σ does not reduce D, then since $D \in F$, $F \wedge \neg C \vdash_1 D$ has a trivial UP refutation. In both cases, we can skip D and go to the next iteration of the loop.

That leaves the reduced clauses $D_{|\sigma}$ to be checked. Since τ stores the unit clauses from UP on $F \wedge \neg C$, it suffices to show that $F \wedge \tau \vdash_1 D_{|\sigma}$. If $\tau \vDash D_{|\sigma}$ (Line 8), then we have a trivial UP refutation. Otherwise, we perform UP with the addition of $\neg D_{|\sigma}$ to τ, forming τ' (Lines 9–13). If UP fails to find a refutation, then we cannot prove that C is SR (Line 14).

There are two potential computational bottlenecks in this algorithm. The first is performing UP. In the worst case, we must reduce every clause in F under τ when looking for unit and empty clauses. Data structures called *watch pointers* [21] are commonly used to efficiently implement this search process. Yet even with watch pointers, a significant amount of SR proof checking time is spent performing UP. One way of making UP more efficient is to be told the series of clauses in F that become unit or empty. And indeed, the hints in the hinted SR proof format are precisely these clauses.

The second bottleneck is creating τ' on Line 9. If τ contains many unit clauses, then we want to avoid copying them into a new τ' object for each loop. One idea is to keep a record of the unit clauses encountered by UP on Lines 10–13, and then undo their effects on τ' afterwards to restore τ. However, this would take time linear in the number of unit clauses. Instead, we want a data structure that can do this in constant time. Such a data structure exists and is commonly used in SAT proof checking tools. We implement it in our SR proof checkers, and we describe how we formalized it in Section VII.

$$
\begin{aligned}
\text{proof} &::= [\text{line}] \\
\text{line} &::= \text{id}, (\text{add} \mid \text{delete}), \texttt{0}, \texttt{\textbackslash n} \\
\text{add} &::= \text{clause}, \langle\text{witness}\rangle, \texttt{0}, [\text{id}], [-\text{id}, [\text{id}]] \\
\text{witness} &::= p : \text{lit}, [\text{lit}], \langle \boldsymbol{p}, \mathbf{[(var, lit)]} \rangle \\
\text{delete} &::= \texttt{d}, [\text{id}] \qquad\qquad \text{clause} ::= [\text{lit}] \\
\text{id}, \text{var} &::= \mathbb{N} \setminus \{0\} \qquad\qquad \text{lit} ::= \mathbb{Z} \setminus \{0\}
\end{aligned}
$$

Fig. 3. The formal grammar for the LSR proof format. A list with 0 or more items is written as $[\cdot]$, and an optional object is written as $\langle\cdot\rangle$. Additions to the LSR format from LPR are bolded.

V. The SR proof formats

We introduce the proof formats for SR without and with hints, which we call DSR and LSR, respectively.[4] Our formats extend the DPR/LPR proof formats [13], which themselves extend DRAT/LRAT [22, 23]. Our formats are backwards compatible, meaning that any RAT or PR proof is also a valid SR proof.

As with RAT and PR, SR proofs comprise *addition* and *deletion* lines. Each addition line contains a clause C claimed to be redundant. If C is nonempty, then the line may also contain a substitution witness σ satisfying C. If no witness is provided, then σ is defined as $\sigma(p) = \top$, where p is the first literal of C (called the *pivot*), and is the identity everywhere else. That way, σ satisfies C.

In the DSR format, addition lines contain only C and σ, and the checker must run Algorithm 1. But in the LSR format, addition lines also contain *hints*. In LSR, each clause is given a unique numerical identifier starting at 1. Each hint is the ID of a clause that becomes unit or conflict under UP. A list of UP refutation hints are provided for each $D_{|\sigma} \in F_{|\sigma}$. While hints do increase the size of the proof, they generally reduce proof checking times by making UP much more efficient.

Deletion lines specify clauses in F to delete. Deletion does not maintain equisatisfiability, but if the formula after deletion from F is unsatisfiable, then so is F. Deletion speeds up proof checking by reducing the required number of $D_{|\sigma}$ UP refutations to find. It can also increase the set of SR clauses, as it can remove $D_{|\sigma}$ clauses that fail the SR check.

The formal grammar for the LSR format is shown in Figure 3. Figure 4 shows a DIMACS CNF formula and its corresponding LSR proof for PHP with $n = 4$. The parts of an LSR addition line are color-coded. Their order is as follows:

1) A (positive, unique) numerical clause ID. Later LSR lines refer to the added clause by this ID.
2) The (literals of the) candidate redundant clause C.
3) An optional substitution witness σ beginning with p, the first literal of C (the pivot). The witness has two parts,

[4] The names are acronyms. DSR stands for "deletion SR," while LSR stands for "linear SR," where "linear" means that the hints included in the format enable proof checking to take time linear in the size of the proof.

CNF format

```
p cnf 12 22
 1  2  3  0
 4  5  6  0
 7  8  9  0
    ...
```

LSR format

```
23 -10 -10 7 -10 8 11 11 8 9 12 12 9 0  7  9 10 -4 3 -6 -8 -12 13 ... 0
24  -7  -7 4  -7 5  8  8 5 6  9  9 6 0 23  6  8 -3 2 -5 -9 -11 12 ... 0
25  -4  -4 1  -4 2  5  5 2 3  6  6 3 0 23 24  5 -2 1 -6 -7 -11 11 ... 0
26 -11                               0 24 25 15 16 2  3 20            0
27                                   0 23 24 25 26 4 21 22   2  3  14 0
```

Fig. 4. A DIMACS CNF formula (left) and its LSR proof (right). Each line in the CNF is a new clause. LSR addition lines comprise a clause ID (pink), the literals of the clause (green), and an optional substitution witness containing literals mapped to true (orange) and variables mapped to other literals (blue), with another appearance of the pivot literal (black) acting as a separator. The line concludes with UP hints (purple) and reduced-clause UP hints (red). Each reduced clause $D_{|\sigma}$ is identified with a negative ID, and the positive hints that follow are the UP refutation for $D_{|\sigma}$.

separated by another appearance of p:[5] first, a list of literals ℓ that σ maps to true (including p); and second, a list of variable-literal pairs (v, ℓ) setting $\sigma(v) := \ell$. All other variables v are mapped to themselves, i.e., $\sigma(v) := v$.

4) A separating `0`, marking where the hints begin.
5) A list of hints, not necessarily deriving UP refutation, guiding Line 2 of Algorithm 1.
6) A list of hints deriving UP refutation for each reduced clause, guiding Line 10. The reduced clause is identified by the negative of its ID, and the UP hints follow.

If each of the line ID, the hints, and the ending `0` are removed, then the addition line becomes a DSR addition line.

Currently, our checkers assume that the witness σ satisfies the candidate clause C (which is the case for all of the SR proofs that appear in this paper). However, the DSR and LSR formats can also express proofs where σ causes $C_{|\sigma}$ to be a tautology: the proof can simply map the pivot p to itself or to any other literal in the substitution portion. We plan to support this general case in the future.

LSR deletion lines start with a line ID, followed by a `d` and the IDs of clauses to be deleted. Historically, the line ID matches the ID of the most-recently-added clause so that unordered proofs can be sorted. But most modern proof-logging SAT solvers emit ordered proofs, so the line ID is ignored.

DSR does not use clause IDs, so deletion lines specify the (literals of the) clause to be deleted directly. Thus, DSR deletion lines only delete a single clause at a time.

The only addition to the LSR format from LPR is the second part of the substitution. If no variable-literal mappings are provided, then the substitution is a partial truth assignment, which is the type of witness used for PR proofs. It is in this way that the SR formats are backwards compatible.

VI. SR Proof Checking Tools

Absent a dedicated SR proof checker, DSR proofs can be checked by converting them into DRAT and then using conventional DRAT/LRAT checkers. We implemented such a conversion algorithm by extending one that converts DPR proofs into DRAT [20]. The most important change to the algorithm is that it uses *multiple* auxiliary variables to convert a single SR clause addition step into a sequence of DRAT proof steps. More specifically, it introduces a fresh variable (i.e., one not appearing in either the formula or the proof) for each SR addition step and several fresh variables for each variable-literal mapping in the substitution.

Our implementation is open-source and can be found at:

`https://github.com/marijnheule/sr2drat.`

We also developed SR proof checking tools. Following the tradition of drat-trim[6] [22] and dpr-trim[7] [13], we present dsr-trim, a tool for adding hints to DSR proofs to create LSR proofs. The source code is available at:

`https://github.com/ccodel/dsr-trim.`

At the moment, dsr-trim can only perform *forwards checking*, which means that it checks DSR proofs from start to finish and adds hints as it goes. In contrast, drat-trim and dpr-trim both additionally implement *backwards checking*, meaning that they read the entire DSR proof into memory first and then work backwards from the derivation of the empty clause, ignoring unreferenced proof lines. In practice, backwards checking can significantly reduce the size of proofs. Adding backwards checking to dsr-trim is ongoing work.

In addition to dsr-trim, we implemented lsr-check, an unverified LSR proof checker. Despite SR being more complicated than PR/DRAT, lsr-check performs comparably to, and often better than, its sister checkers lrat-check and lpr-check.

Both dsr-trim and lsr-check are configured to parse and/or produce proofs in the ASCII format presented in Figure 4 and in a custom binary format that is faster to parse. Compressed proofs tend to be 50-60% smaller in file size. The `compress/decompress` tools included with dsr-trim translate DSR and LSR proofs into and out of this binary format.

VII. The Verified LSR Checker

In this section, we discuss our implementation and verification of an LSR proof checker in the Lean 4 interactive theorem prover [12]. Our checker is open-source and can be found at:

`https://github.com/FormalSAT/trestle.`

[5] The choice to use p as a separator is a historical one. In PR, the pivot appears twice: once in the clause, and once to indicate when the partial assignment begins. For SR, we need a way to indicate when variable-literal mappings occur. Since `0` is a reserved symbol, we use p once again.

[6] `https://github.com/marijnheule/drat-trim.`
[7] `https://github.com/marijnheule/dpr-trim.`

The core of the checker is a function called `checkLine` that runs Algorithm 1 with UP hints (i.e., LSR). Ideally, we would expect its correctness theorem to look like this:

```
checkLine F C line = ok → eqsat F (F ∧ C).
```

And indeed, this is equivalent to what we proved in Lean. But to make our checker efficient, we implemented data structures that cause the correctness theorem to look more complicated:

```
theorem checkLine_ok : models R F C →
  checkLine ⟨R, τ, σ⟩ line = .ok S →
  eqsat F (F ∧ C) := by ...
```

There are three main differences between the correctness theorems. The first is the use of a functional programming idiom similar to the state monad: `checkLine` takes a state triple of a CNF data structure `R`, a partial assignment τ, and a substitution σ, along with the LSR `line`, and it returns an updated $\langle \mathtt{R}, \tau, \sigma \rangle$ state triple `S` and its yes/no result. By passing along `R`, τ, and σ, Lean can allocate the memory for these structures once, rather than at the start of each proof line, which makes the checker more efficient.

The second difference is that the literals of the candidate clause C are stored in `R` rather than in a separate object. This eliminates the need of writing the literals twice: once during parsing, and once when copying them into `R` after checking that C is SR. According to a CPU profiler, this sleight of hand gives a 10% speedup on our longest-running benchmark.

The third and greatest difference is how we implement `R`, τ, and σ. We implement CNF formulas with a data structure we call `RangeArray`. In the correctness theorem, we assume that `R models` formula F and candidate clause C. We implement τ and σ with data structures we call `PPA` and `PS`, standing for "persistent partial assignment" and "persistent substitution." These three data structures enable our checker to efficiently implement UP, but at the cost of a much more complicated proof of correctness.

In total, our verified checker and its supporting theorems and data structures comprise 8k LoC, and the verification took 4 person months. Much of that time was spent adjusting how the checker iterates across data structures in order to make the compiled Lean code performant. For example, implementing reduction (i.e., $C_{|\sigma}$) with an API-breaking, tail-recursive function, as opposed to a `foldlM` in the `Except` monad, gave an immediate speedup of 60% on our longest-running benchmark. We hope that future versions of the Lean compiler will be less picky about generating performant code.

In the rest of this section, we discuss the `RangeArray` and `PPA` data structures, as they represent the most technical portions of our verification. These data structures use techniques common in other SAT solving tools, including dsr-trim.

A. RangeArray

Given a type of boolean literals `ILit`, a straightforward type for CNF formulas is `List (List ILit)`. We use this datatype in our SAT theory, since Lean provides good support for lists. However, this datatype suffers from two drawbacks. The practical drawback is that a nested list unnecessarily fragments the memory for clauses across separately-allocated blocks, leading to additional memory overhead and reduced cache locality. The other drawback is that nested lists cannot easily implement clause deletion. Recall that SR proofs may delete clauses from the formula. Because LSR hints are static IDs, we cannot simply remove deleted clauses from the nested list, as this would shift the indexes of the remaining clauses.

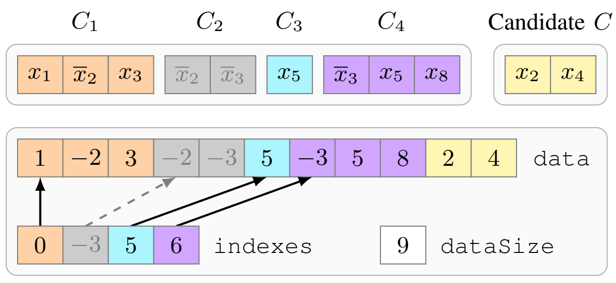


Fig. 5. An example of a `RangeArray` modeling a formula with four clauses and a candidate clause C. All literals are stored in a single array `data`, and clauses are defined based on index "pointers" in `indexes`. The candidate clause is implicitly defined as being the additional literals in `data` beyond the index stored in `dataSize`. The `RangeArray` deletes clauses by marking their index as negative. In the example, clause C_2 is deleted.

One hack is to replace the deleted clause with the binary clause $C_\top := x_1 \vee \overline{x}_1$, since a tautology behaves like $\top$ in our SAT theory. But then the proof checker would be hard-coded to check for deletion by comparing clauses to $C_\top$, which strikes us as inelegant and non-modular. Another solution is to use option types.[8] However, options (in our opinion) clutter up code and proofs, and they add another layer of pointer indirection in compiled code, which leads to slower runtimes.

Instead, we implemented a common data structure for CNF formulas we call `RangeArray`. Figure 5 shows an example. `RangeArrays` flatten the nested list datatype so that all literals lie in a single array `data`, and clauses are defined using index "pointers" stored in a second array `indexes`. Intuitively, the ith clause starts at position `indexes[i]` in `data`, and it has size `indexes[i+1] - indexes[i]`. However, deletion is implemented by setting an index p to $-p$, so calculations involving indexes use their absolute value.

The `RangeArray` has two main benefits. The first benefit is that all literals lie in the same array, so iteration across an entire formula has increased cache locality. The second benefit is the ability to store the candidate clause C in the `RangeArray` during proof checking. We do so by adding the literals of C to `data` without assigning C an index. To differentiate between formula literals and candidate clause literals, we store the total number of formula literals in a variable `dataSize`. Thus, the literals of C lie between `dataSize` and the actual size of `data`. The `commit` operation adds C to the formula by assigning C an index and increasing `dataSize` by $|C|$.

We relate `RangeArray` to our model for CNF formulas and clauses with the `models` predicate. Given a formula `F` and a candidate clause `C`, `models R F C` means that

[8] Values of type `Option V` are either `none` or `some v`, where `v:V`.

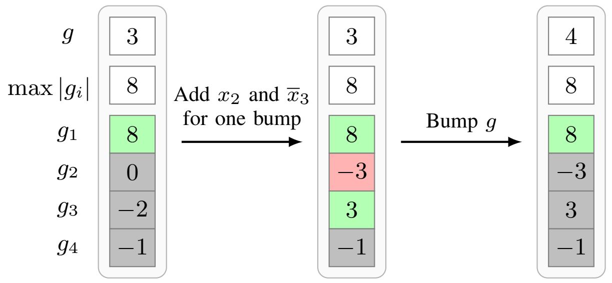


Fig. 6. An example of the PPA data structure in use. If $|g_i| \geq g$, then the sign of g_i determines if $\tau(x_i)$ is true (green) or false (red). Otherwise, $\tau(x_i)$ is undefined (gray). On the left, $\tau := x_1$, and its truth value is set for 6 rounds of UP (6 bumps). In the middle, the unit clauses x_2 and $\overline{x}_3$ are added to τ. On the right, those two truth values are cleared with a bump in $O(1)$ time.

`R` agrees with `F` and `C` on every non-deleted clause, such that `R[indexes[i]+j] = F[i][j]` and `R[dataSize+i] = C[i]`. We prove that `commit` and other operations preserve the `models` predicate in the appropriate way.

B. *Persistent partial assignments*

Partial truth assignments can be implemented with an `Array (Option Bool)`. For any variable v, if `A[v] = none`, then $\tau(v)$ is undefined, and otherwise `A[v] = some b` means that $\tau(v) = b$. However, this implementation would make it inefficient to run Lines 1 and 9 of Algorithm 1: an array of booleans would need to be cleared for each LSR line, and the array would need to be restored from τ' to τ after each reduced-clause check, of which there might be many. All of this copying and clearing would make proof checking intractable.

A common solution to this problem is to use a technique we call *generation bumping* (or *timestamping*), which enables $O(1)$ clearing of UP unit clauses. The idea is for the PPA to store a global *generation number* g and a generation number g_i for each boolean variable x_i. If $|g_i| \geq g$, then the sign of g_i determines if $\tau(x_i)$ is true ($+$) or false ($-$). Otherwise, $\tau(x_i)$ is undefined. Incrementing g, called *bumping*, clears the truth values of any x_i with $|g_i| = g$. Setting $g := \max|g_i| + 1$, called *clearing*, clears all truth values.

Proof checkers can use generation bumping because they know in advance the exact number of bumps any particular truth value should be set for. Unit clauses found during UP in the entailment phase (Lines 1–4) must persist in τ for all rounds of UP in the SR phase (Lines 10–13). Since each reduced clause UP refutation is marked in the LSR line, proof checkers can count the expected number of refutations r during parsing, and then set the generation number for unit clauses in τ to $|g_i| := g + r + 1$. In the SR phase, new unit clauses added to τ' have their generation numbers set to g so that they can be cleared afterwards with a single bump. Figure 6 illustrates how this works. Our verification of `checkLine` includes careful bookkeeping to ensure that the unit clauses in τ persist.

Our implementation of PS also uses generation bumping, except that an additional array stores what each variable is mapped to under the substitution.

VIII. Experimental results

Our experimental results demonstrate the clear benefits of using a strong proof system. We highlight three main results: (1) that our verified LSR checker performs well against cake_lpr [13], a fast, verified LPR checker, (2) that SR proofs are smaller than their RAT counterparts, and (3) that our verified checker incurs reasonable overhead compared to lsr-check, our unverified LSR checker written in C.

Our benchmarks comprise five families of SR formulas: Ramsey instance $R(4,4) \leq 18$, Schur number five [24], a packing problem [1], and PHP and Tseitin formulas. We also include a similar set of five PR families, where instead of packing, Schur, and Ramsey, it has Mycielski [25], mutilated chessboard, and two-pigeons-per-hole (tph) formulas [26].

We ran our experiments on a 2022 M1 Mac Studio with 32 GB of memory and a clock speed of 3.2 GHz. To replicate our results, use the scripts found at:

`https://github.com/ccodel/sr-benchmarking`.

A. *Comparison to cake_lpr*

We first show that our verified Lean checker performs similarly to cake_lpr. Our experiments covered the PR proof families. Figure 7 summarizes our results.

For proofs that took longer than 1 second to verify, our checker took an average of 81.95 seconds, while cake_lpr took an average of 124.86 seconds. The geometric mean of the ratio of Lean / cake_lpr runtimes was 0.718.

For proofs that took less than 1 second to verify, our checker took proportionally longer than cake_lpr. For example, the geometric mean of the ratios of runtimes on these instances was 5.84. A CPU profiler revealed that 75% of the runtime on these instances was spent on Lean's initialization code that is run only once at the start of the program, and so this does not represent a bottleneck for larger proofs.

B. *Comparison of LSR and converted LRAT proofs*

Next, we show that SR proofs are smaller than their DRAT counterparts. For these experiments, we used sr2drat to translate the DSR proofs into DRAT, which were then translated into LRAT by drat-trim. (We convert from SR to DRAT instead of SR to PR because we do not know of a way to use the PR rule when converting from SR.)

The SR proofs are indeed smaller, both in terms of file size and the number of proof lines. Figure 8 shows our results. On average, an LSR proof was 6.2 MB and had 13.2K proof lines, while the translated LRAT proof was 41.2 MB and had 1.03M proof lines. The geometric means of ratios of LSR to LRAT for file size and line count were 0.085 and 0.004, respectively.

Unsurprisingly, the smaller SR proofs were faster to check. Figure 9 shows the runtimes for our Lean checker on the LSR proofs compared to cake_lpr on the converted LRAT proofs. On average, our checker took 1.06 seconds, while cake_lpr took 4.00 seconds. For proofs that took longer than 1 second to check, the geometric mean of the ratio of Lean / cake_lpr proof checking times was 0.255.

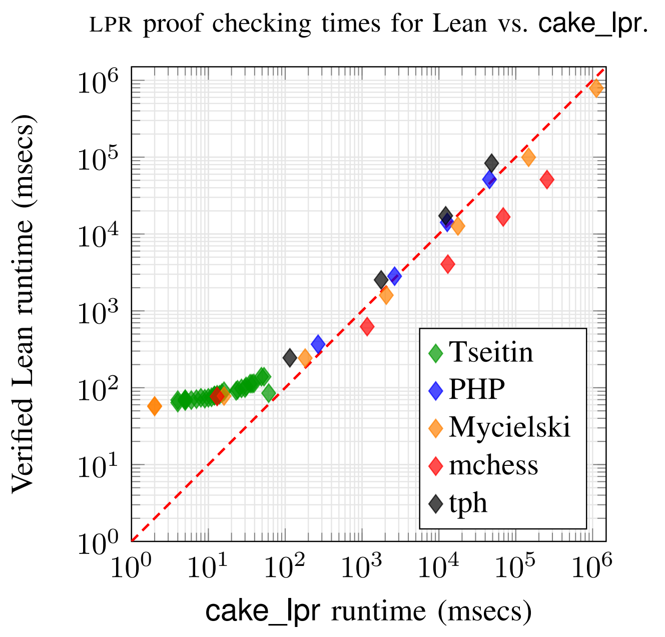


Fig. 7. Comparison of proof checking times for our Lean checker and cake_lpr on the PR proof families. Points below the red $y = x$ line indicate that our checker was faster than cake_lpr.

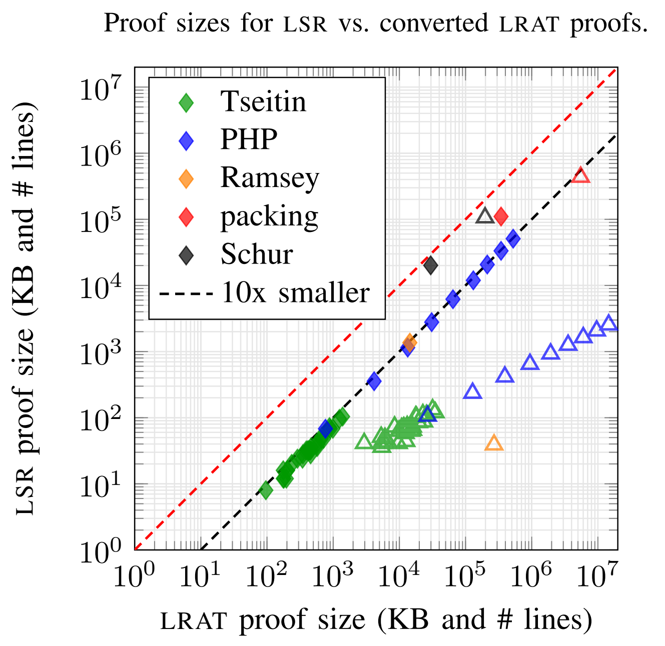


Fig. 8. Proof sizes in terms of KB (♦) and number of proof lines (△) for LSR proofs and their LRAT conversions via sr2drat and drat-trim. Points below the red $y = x$ line indicate that the SR proofs were smaller.

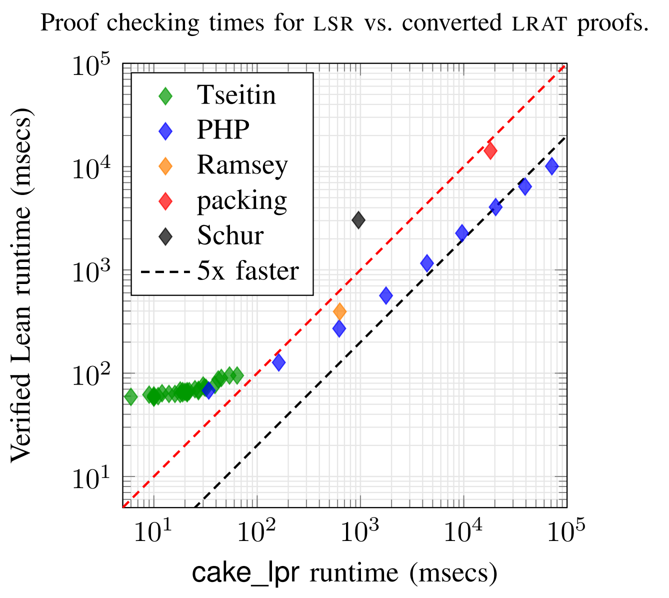


Fig. 9. Comparison of proof checking times for our Lean checker on the LSR proofs and for cake_lpr on the LRAT conversions. Points below the red $y = x$ line indicate that our checker was faster than cake_lpr.

C. *Comparison of verified and unverified checking*

Finally, we report that the added constant factors of our verified proof checker are not excessive. On the LSR proofs, our Lean checker is about 10x slower than our unverified checker lsr-check. Figure 10 summarizes our results. On average, our Lean checker took 1.06 seconds, while lsr-check took 0.10 seconds. For proofs that took longer than 1 second to check, the geometric mean of the ratios of their runtimes was 9.936.

IX. Conclusion and Future Work

In this paper, we presented our tools for verified SR proof checking. SR admits short proofs for many problems, and our experimental results show the clear advantages of using SR over weaker proof systems such as RAT and PR. While no modern SAT solver supports SR reasoning yet, we hope that our tools—including our verified SR proof checker—will support the future development of SR tooling for SAT solving.

There are several avenues for future work. One such avenue is improving dsr-trim and our verified Lean checker. We plan to add backwards proof checking to dsr-trim. In addition, the Lean checker can be made more efficient by minimizing the number of clauses it checks. We have seen that if $D_{|\sigma} = D$, then it can be skipped during the SR phase. By storing the first and last clause containing each literal, we can compute the range of clauses reduced by σ such that clauses outside of this range are not reduced, and thus do not need to be checked. Our lsr-check tool implements this technique. While it only gives modest speedups, we hope that implementing it in Lean will improve the Lean checker's runtime.

Another avenue of future work is in automatically generating symmetry-breaking SR proofs. CDCL SAT solvers tend to struggle on problems with a high degree of symmetry. Adding

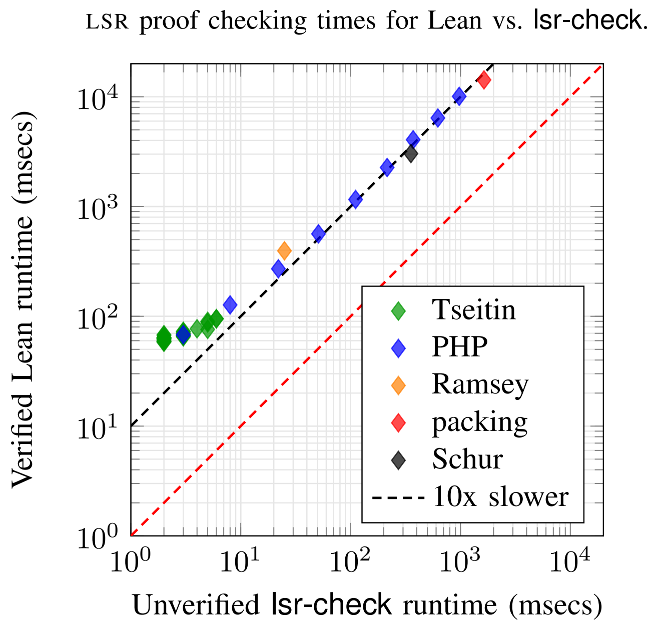


Fig. 10. Comparison of proof checking times for our Lean checker against our unverified checker lsr-check on the SR proof families. Marks on the black dashed line indicate that the Lean checker was 10x slower than lsr-check.

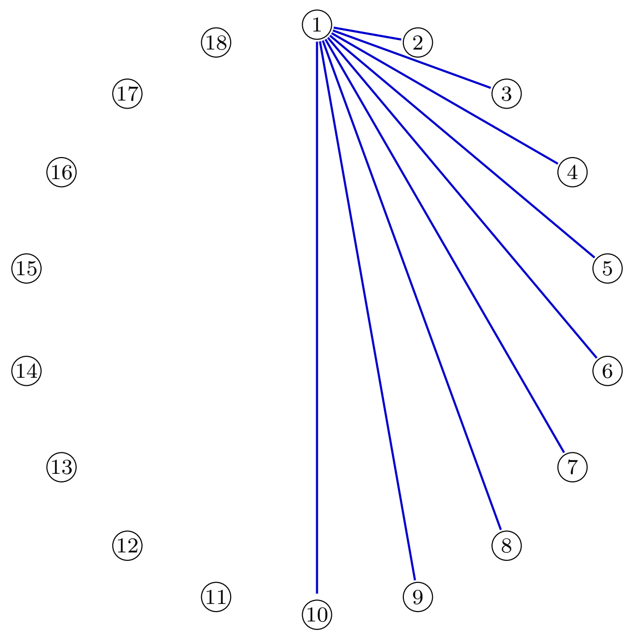


Fig. 11. First phase: sort the edges for vertex v_1 so the blue edges come first, and fix the edge $e_{1,10}$ to blue.

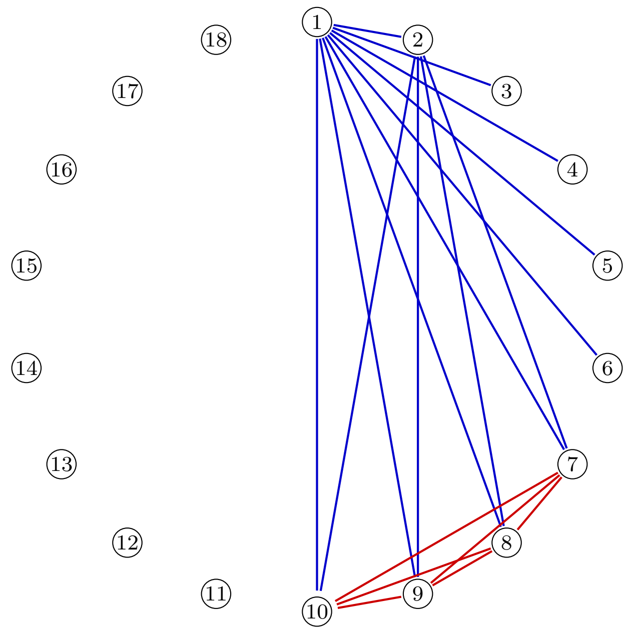


Fig. 12. Second phase: sort the edges for vertex v_2 so the red edges come first, and fix $e_{2,7}$ to blue. This results in a conflict via unit propagation: a red 4-clique v_7, v_8, v_9, v_{10}. As a consequence we can fix $e_{2,7}$ to red.

symmetry-breaking clauses via SR proof steps can be a trusted way to improve solver runtimes.

X. Short SR Proof of $R(4,4) \leq 18$

We constructed a short SR proof of $R(4,4) \leq 18$ that consists of only 38 clauses. The proof consists of four phases. In the first phase, we assume WLOG that vertex v_1 is connected to at least nine blue edges and that these blue edges connect v_1 to the vertices $v_2, \ldots, v_{10}$. To show this, we sort the edges adjacent to vertex v_1 so that the blue edges appear first. The clauses that express the sorting are of the form $e_{1,i} \vee \bar{e}_{1,i+1}$ with $1 < i < n$. The SR witnesses for these clauses are a permutation of the vertices. At this point, we can still exchange the two colors. We use this to fix the edge $e_{1,10}$ to blue. The result is shown in Figure 11. This phase consists of 17 clause addition steps.

In the second phase, we assume WLOG that v_2 is connected to at least five red edges and that these red edges connect v_2 to the vertices $v_3, \ldots, v_7$. In the proof, we sort the edges adjacent to vertex v_2 with the red edges appearing before the blue edges. The clauses that express the sorting are of the form $\bar{e}_{2,i} \vee e_{2,i+1}$ with $2 < i < 11$. If we assign edge $e_{2,7}$ to blue, then unit propagation will result in a conflict as shown in Figure 12. Thus, we may fix $e_{2,7}$, $e_{2,6}$, $e_{2,5}$, $e_{2,4}$, and $e_{2,3}$ to red. This phase consists of 9 clause addition steps.

In the third phase, observe that there cannot be a red or blue 3-clique among the vertices v_3, v_4, v_5, v_6, and v_7, because all of them are connected to v_1 with a blue edge and all of them are connected to v_2 with a red edge. There is a unique red-blue assignment (modulo symmetry) that avoids a red or blue 3-clique among five vertices: a blue 5-cycle and a red 5-cycle. We fix this assignment after sorting the edge for vertex v_3 (blue edges first). The result is shown in Figure 13. This phase consists of 7 clause addition steps.

In the fourth and final phase, we first determine that the edges $e_{2,8}$, $e_{2,9}$, and $e_{2,10}$ must be blue. This is achieved with two clauses. The first clause assumes $e_{2,8}$ and $e_{3,8}$ are red. This results in a conflict by unit propagation. Afterwards we only assume that $e_{2,8}$ is red. This now results in a conflict by unit propagation as well, as $e_{3,8}$ is forced to be blue. The

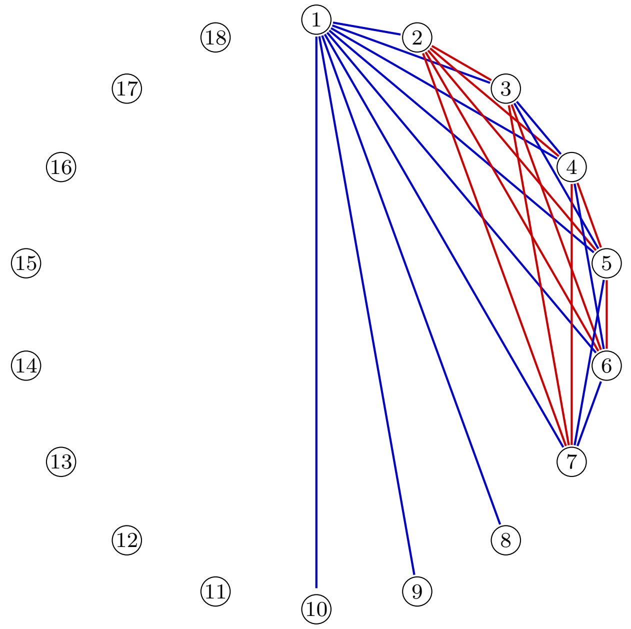


Fig. 13. Third phase: There cannot be a 3-clique in red nor a 3-clique in blue among v_3, v_4, v_5, v_6, and v_7. There is a unique assignment that achieve this. We sort the edges among these vertices and fix that unique assignment.

failed assumption allows us to fix $e_{2,8}$ to blue. See Figure 14.

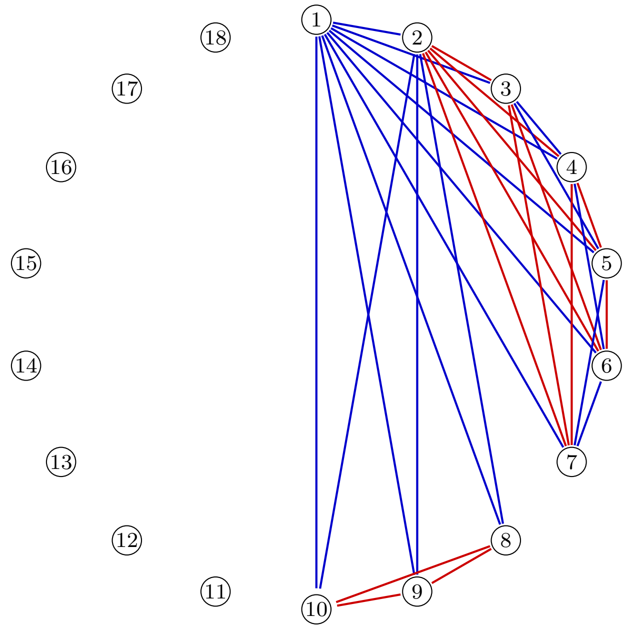


Fig. 14. Fourth phase: determine that $e_{2,8}$, $e_{2,9}$, and $e_{2,10}$ must be blue.

Afterward the edges $e_{3,8}$, $e_{3,9}$, and $e_{3,10}$ are sorted (red first). This step is allowed because vertices v_8, v_9, and v_{10} are still interchangeable at this point. Assuming that $e_{3,9}$ is blue results in a conflict by UP, so $e_{3,9}$ (and thus $e_{3,8}$) needs to be red. The final refutation comes from the observation that assuming either $e_{4,8}$ to red or blue results in a conflict by UP. This phase consists of 7 clause addition steps.

REFERENCES

[1] B. Subercaseaux and M. J. H. Heule, "The packing chromatic number of the infinite square grid is 15," in *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pp. 389–406, 2023.

[2] M. J. H. Heule and M. Scheucher, "Happy ending: An empty hexagon in every set of 30 points," 2024.

[3] Z. Li, C. Bright, and V. Ganesh, "A SAT solver and computer algebra attack on the minimum kochen-specker problem," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 23559–23560, Mar. 2024.

[4] N. Rungta, "A billion SMT queries a day (invited paper)," in *Computer Aided Verification*, pp. 3–18, 2022.

[5] M. Järvisalo, M. J. H. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning*, pp. 355–370, 2012.

[6] M. J. H. Heule, B. Kiesl, and A. Biere, "Strong extension-free proof systems," *Journal of Automated Reasoning*, vol. 64, no. 3, pp. 533–554, 2020.

[7] *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B, 2023.

[8] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, "Preprocessing of propagation redundant clauses," *Journal of Automated Reasoning*, vol. 67, Sep 2023.

[9] S. Buss and N. Thapen, "DRAT and propagation redundancy proofs without new variables," *Logical Methods in Computer Science*, vol. Volume 17, Issue 2, Apr 2021.

[10] S. Gocht and J. Nordström, "Certifying parity reasoning efficiently using pseudo-boolean proofs," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 3768–3777, May 2021.

[11] A. Rebola-Pardo, "Even Shorter Proofs Without New Variables," in *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)* (M. Mahajan and F. Slivovsky, eds.), vol. 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 22:1–22:20, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[12] L. d. Moura and S. Ullrich, "The Lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28*, pp. 625–635, 2021.

[13] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "Verified propagation redundancy and compositional UNSAT checking in CakeML," *International Journal on Software Tools for Technology Transfer*, vol. 25, no. 2, pp. 167–184, 2023.

[14] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Principles of Programming Languages (POPL)*, pp. 179–191, Jan 2014.

[15] M. J. H. Heule, *Proofs of Unsatisfiability*, ch. 15, pp. 635–668. Frontiers in Artificial Intelligence and Applications, 2 ed., 2021.

[16] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere,

"PRuning through satisfaction," in *Hardware and Software: Verification and Testing*, pp. 179–194, 2017.

[17] A. Haken, "The intractability of resolution," *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[18] S. A. Cook, "A short proof of the pigeon hole principle using extended resolution," *SIGACT News*, vol. 8, pp. 28–32, oct 1976.

[19] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, pp. 466–483. 1983.

[20] M. J. H. Heule and A. Biere, "What a difference a variable makes," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 75–92, 2018.

[21] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference*, pp. 530–535, 2001.

[22] N. Wetzler, M. J. H. Heule, and W. A. Hunt, "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Theory and Applications of Satisfiability Testing – SAT 2014*, pp. 422–429, 2014.

[23] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction – CADE 26*, pp. 220–236, 2017.

[24] M. J. H. Heule, "Schur number five," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI'18, 2018.

[25] E. Yolcu, X. Wu, and M. J. H. Heule, "Mycielski graphs and PR proofs," in *Proceedings of the 23rd Conference on Theory and Applications of Satisfiability Testing (SAT)*, no. 12178 in Lecture Notes in Computer Science, pp. 201–217, 2020.

[26] A. Biere, "Two pigeons per hole problem," in *Proc. of SAT Competition 2013: Solver and Benchmark Descriptions*, p. 103, 2013.

2-DQBF Solving and Certification via Property-Directed Reachability Analysis

Long-Hin Fung*, Che Cheng*, Yu-Wei Fan*, Tony Tan†, Jie-Hong Roland Jiang*
*National Taiwan University, Taipei, Taiwan
{r12922017, r11943097, r11943096, jhjiang}@ntu.edu.tw
†University of Liverpool, Liverpool, England
tonytan@liverpool.ac.uk

***Abstract*—Recently a refined complexity analysis of the satisfiability of Dependency Quantified Boolean Formula (DQBF) was established. In particular, it is shown that the satisfiability of 3-DQBF (i.e., DQBF with 3 existential variables) is NEXP-complete and it becomes PSPACE-complete for 2-DQBF. While all state of the art DQBF solvers focus on general DQBF, it is natural to ask if there is an efficient approach for solving 2-DQBF – similar to how modern SAT solvers differentiate between 2-SAT and 3-SAT instances.**

In this paper we show how to exploit modern Property Directed Reachbility (PDR) solvers to solve 2-DQBF instances. We present a novel linear time reduction from 2-DQBF instances to PDR instances and show how to convert the inductive-invariant certificates provided by PDR solvers to the Skolem-function certificates for 2-DQBF instances. The experimental results show that the approach is indeed more efficient than other state-of-the-art DQBF solvers, at least in solving 2-DQBF instances.

***Index Terms*—Dependency quantified Boolean formula (DQBF), property directed reachability (PDR), model extraction, Skolem functions**

I. Introduction

The dependency quantified Boolean formula (DQBF) [1, 2] extends the quantified Boolean formula (QBF) [3] with Henkin quantifiers [4], which allow the dependency set of an existential variable to be explicitly specified. This extension makes DQBF a natural formalism for important applications in system synthesis and verification, such as black-box synthesis [5, 6], controller synthesis [7], engineering change order [8], distributed synthesis for LTL fragments [9], etc, all of which are beyond the expressiveness of QBF. These amotivated the development of various DQBF solvers, e.g., `HQS` [10], `Pedant` [11, 12], `DQBDD` [13]. However, the extension also lifts the complexity of the satisfiability for DQBF to NEXPTIME-complete [1], in contrast to QBF which is "only" PSPACE-complete.

Recent theoretical advancement uncovers important properties of special sub-classes of DQBF. For example, the satisfiability of DQBF^{de} – DQBF with disjoint or equal dependency sets and CNF matrix – is shown to be in PSPACE [14]. Recently in [15] it is shown that the complexity of DQBF depends on the number of existential variables in the same way as the complexity of SAT depends on the width of the clauses. For example, the complexity of 2-DQBF, i.e., DQBF with 2 existential variables, is PSPACE-complete and for 3-DQBF, it becomes NEXP-complete. This is analogous to SAT whose complexity is NL-complete and NP-complete for 2-SAT and 3-SAT, respectively. The main difference is the exponential blow-up for the DQBF counterpart.

Analogous to modern SAT solvers that often differentiate between 2-SAT and 3-SAT instances and solve them using different methods, it is natural to ask whether we can do the same for DQBF. In this paper, we investigate this question. We are going to exploit the fact that 2-DQBF is essentially a succinct version of 2-CNF formula [15], which implies that the (un)satisfiability of 2-DQBF can be established by checking whether there is a cycle in the (implicit) implication graph that contains two vertices whose labeling assignments are contradicting. We transform such a cycle detection problem into a reachability problem in a way similar to the liveness-to-safety conversion [16]. This conversion allows the state-of-the-art model checking algorithms, such as IC3 [17], Property-Directed Reachability (PDR) [18], and Abstractly Verifying Reachability (AVR) [19], to be exploited for 2-DQBF solving.[*]

We also show how to convert the certificates provided by the PDR algorithm to the certificates for 2-DQBF. For a false 2-DQBF instance, the trace of the corresponding reachability can be converted directly to a certificate that witnesses the falsity. For a true 2-DQBF instance, however, the conversion is not as straightforward. In this case, the PDR algorithm gives us an inductive state set separating the initial states from the final states derived from the PDR computation and this set is only an over-approximation of the reachable states. We will show how to progressively refine the transition system until it outputs an inductive set that reflects the appropriate Skolem functions for the original 2-DQBF instance.

Note that at first glance, 2-DQBF and PDR may seem to have little in common. PDR is essentially about graph reachability problem, while 2-DQBF is about circuits with two black-boxes. Thus, in retrospect, it is surprising that 2-DQBF can be solved using PDR, owing to the connection between 2-DQBF and 2-SAT established in [15].

Experimental results show that our approach outperforms all state-of-the-art DQBF solvers which demonstrates the effectiveness of our approach, at least in solving 2-DQBF instances. We also compare it with QBF solvers where we

[*]In this paper we refer to the IC3-based model-checking algorithms as PDR algorithms.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_25

reduce 2-DQBF instances to QBF instances. In all instances, the QBF solvers time out. This is not surprising since the only known reduction to QBF is the one in [3], which yields a quadratic blow-up in the number of variables, hence, instances quickly become too large and beyond the capability of even the best QBF solvers.

This paper is organized as follows. In Section II we briefly review the basic notations on DQBF and PDR. We show the reduction from 2-DQBF instances to PDR instances as well as the Skolem functions extraction in Section III. Our experimental results are presented in Section IV. We conclude with Section V. Our code and benchmarks are publicly available on https://github.com/LH104729/2-DQBF-Solving-and-Certification-via-Property-Directed-Reachability-Analysis.

II. Preliminaries

Let $\Sigma = \{0, 1\}$, where 0 and 1 represent the Boolean values *false* and *true*, respectively. As usual, $\neg 0 = 1$ and $\neg 1 = 0$. Let Σ^i and Σ^* denote the sets of Boolean strings of length i and arbitrary length, respectively. We use the symbols a, b, c to denote Boolean constants, i.e., elements in Σ, and the bar version $\bar{a}, \bar{b}, \bar{c}$ to denote Boolean constant vectors, i.e., strings in Σ^* with $|\bar{a}|$ denoting the length of $\bar{a}$. Tuples of values from Σ are written as strings, e.g., 100 denotes $(1, 0, 0)$. Boolean variables are denoted by x, y, z, u, v and the bar version $\bar{x}, \bar{y}, \bar{z}, \bar{u}, \bar{v}$ denote vectors of Boolean variables with $|\bar{x}|$ denoting the length of $\bar{x}$. We insist that in a vector $\bar{x}$ there is no variable occurring more than once. We write $\bar{x}'$ to denote the vector obtained by priming all the variables in $\bar{x}$. For convenience, we view the vectors $\bar{x}, \bar{y}, \bar{z}$ as sets of variables and use set-theoretic operations on them, e.g., $\bar{z} \subseteq \bar{x}$ means every variable in $\bar{z}$ also occurs in $\bar{x}$.

As usual, $\varphi(\bar{x})$ denotes a (Boolean) formula[†] with variables $\bar{x}$. When the variables are not relevant or clear from the context, we simply write φ. For $\varphi(\bar{x})$ and $\psi(\bar{z})$ where $\bar{z} \subseteq \bar{x}$, we write $\varphi(\bar{x}) \Rightarrow \psi(\bar{z})$ to denote that every satisfying assignment of φ is also a satisfying assignment of ψ.

Let $\varphi(\bar{x})$ be a formula. Let $\bar{z} = (z_1, \ldots, z_m) \subseteq \bar{x}$ and $\bar{z}' = (z'_1, \ldots, z'_m)$. We write $\varphi[\bar{z}/\bar{z}']$ to denote the formula obtained by simultaneously substituting each z_i with z'_i for each $1 \leqslant i \leqslant m$. For a string $\bar{a} = (a_1, \ldots, a_m) \in \Sigma^m$, $\varphi[\bar{z}/\bar{a}]$ denotes the formula obtained by assigning each a_i to z_i. When $\bar{z} = \bar{x}$, we just write $\varphi[\bar{a}]$ instead of $\varphi[\bar{x}/\bar{a}]$.

For $\bar{z} \subseteq \bar{x}$ and $\bar{a} \in \Sigma^{|\bar{x}|}$, we write $\bar{a}\big|_{\bar{x}\downarrow\bar{z}}$ to denote the projection of $\bar{a}$ to the components in $\bar{z}$ according to the order of the variables in $\bar{x}$. For example, if $\bar{x} = (x_1, \ldots, x_5)$ and $\bar{z} = (x_1, x_2, x_5)$, then $00101\big|_{\bar{x}\downarrow\bar{z}}$ is 001, i.e., the projection of 00101 to its 1st, 2nd and 5th bits.

A. Dependency Quantified Boolean Formula (DQBF)

A *dependency quantified Boolean formula* (DQBF) in prenex normal form is a formula of the form:

$$\Phi \;:=\; \forall \bar{x}\ \exists y_1(\bar{z}_1) \cdots \ \exists y_k(\bar{z}_k) \quad \phi \tag{1}$$

[†]The results in this paper also hold when a formula is written in circuit form, thus, the term "formula" can be taken to also mean "circuit".

where each $\bar{z}_i \subseteq \bar{x}$ and ϕ, called the *matrix*, is a quantifier-free Boolean formula using variables in $\bar{x} \cup \{y_1, \ldots, y_k\}$. We called $\bar{x}$ the *universal variables*, $y_1, \ldots, y_k$ the *existential variables*, and each $\bar{z}_i$ the *dependency set* of y_i. We call Φ a k-DQBF, where k is the number of existential variables in Φ.

A DQBF Φ in the form of Eq. (1) is *satisfiable* if there is a tuple $(f_1, \ldots, f_k)$ of functions, where $f_i : \Sigma^{|\bar{z}_i|} \to \Sigma$ for every $1 \leqslant i \leqslant k$, and by replacing each y_i with $f_i(\bar{z}_i)$, the formula ϕ becomes a tautology. The tuple $(f_1, \ldots, f_k)$ is called the *(satisfying) Skolem functions* for Φ and we say that Φ is satisfiable by the Skolem functions $(f_1, \ldots, f_k)$, i.e., the Skolem functions form a model of Φ.

It is known that the complexity of the satisfiability problem for DQBF is parametric in k, similar to k-SAT: When $k = 2$, it is PSPACE-complete and when $k = 3$, it becomes NEXP-complete and that there is a parsimonious polynomial-time reduction from general DQBF to 3-DQBF [15].

Even before [15] it is already known that k-DQBF is indeed k-CNF formula in an exponentially more succinct representation [20, 21, 22]. We will briefly review this equivalence achieved by a simple rewriting technique from [15], which will be useful later on. Let Φ be k-DQBF as in Eq. (1).

For each $1 \leqslant i \leqslant k$ and for each $\bar{c} \in \Sigma^{|\bar{z}_i|}$, let $X_{i,\bar{c}}$ be a variable and for $d \in \Sigma$, we define the literal $L^d_{i,\bar{c}}$ as:

$$L^d_{i,\bar{c}} \;:=\; \begin{cases} \neg X_{i,\bar{c}} & \text{if } d = 0 \\ X_{i,\bar{c}} & \text{if } d = 1 \end{cases}$$

Note that $L^d_{i,\bar{c}} = 1$ if and only if $X_{i,\bar{c}} = d$.

For each $(\bar{a}, \bar{b}) \in \Sigma^n \times \Sigma^k$, where $\bar{a} = (a_1, \ldots, a_n)$ and $\bar{b} = (b_1, \ldots, b_k)$, define the clause $C_{\bar{a},\bar{b}}$ as:

$$C_{\bar{a},\bar{b}} \;:=\; L^{\neg b_1}_{1,\bar{c}_1} \vee \cdots \vee L^{\neg b_k}_{k,\bar{c}_k}$$

where $\bar{c}_i = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_i}$, for each $1 \leqslant i \leqslant k$. The expansion of Φ to a k-CNF formula, denoted by $\exp(\Phi)$, is defined as:

$$\exp(\Phi) \quad := \bigwedge_{(\bar{a},\bar{b}) \text{ s.t. } \phi[(\bar{x},\bar{y})/(\bar{a},\bar{b})]=0} C_{\bar{a},\bar{b}}$$

It is known that Φ is satisfiable if and only if its expansion $\exp(\Phi)$ is satisfiable (in the sense of Boolean formula) [15]. Moreover, a satisfying Skolem functions $(f_1, \ldots, f_k)$ for Φ correspond to a satisfying assignment of $\exp(\Phi)$, where $X_{i,\bar{c}} = f_i(\bar{c})$ for every $1 \leqslant i \leqslant k$ and $\bar{c} \in \Sigma^{|\bar{z}_i|}$.

As mentioned in the introduction, one of the main applications of DQBF is black-box synthesis, also known as Partial Equivalence Checking (PEC). It is defined as given a Boolean circuit with some black-boxes, check whether there is an implementation of the black-boxes such that the function of the whole circuit is a tautology. It can be naturally expressed as the satisfiability of DQBF where the number of black-boxes corresponds to the number of existential variables in the DQBF.

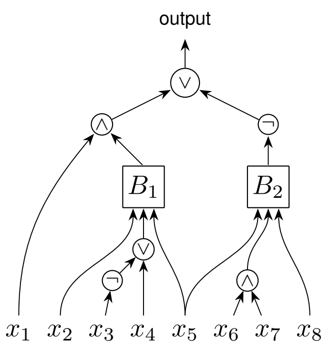


Fig. 1: A PEC example with two black-boxes B_1 and B_2 to be synthesized.

Consider, for example, the circuit with two black-boxes in Figure 1. The PEC solution is equivalent to the Skolem functions of the following 2-DQBF:

$$\forall x_1 \cdots \forall x_{10}\ \exists y_1(x_2, x_9, x_5)\ \exists y_2(x_5, x_{10}, x_8)$$
$$\Big(x_9 = (\neg x_3 \vee x_4) \wedge x_{10} = (x_6 \wedge x_7)\Big) \to \Big((x_1 \wedge y_1) \vee \neg y_2\Big)$$

The additional variables x_9, x_{10} serve as the Tseitin variable representing the values of $\neg x_3 \vee x_4$ and $x_6 \wedge x_7$, respectively, and y_1 and y_2 are the output variables of the two black-boxes with the dependency sets (x_2, x_9, x_5) and (x_5, x_{10}, x_8).

B. Property-Directed Reachability (PDR)

A *finite-state transition system* is a system $\mathcal{S} = (\bar{x}, I, T)$, where $\bar{x}$ is a finite set of Boolean variables, the initial condition $I(\bar{x})$ is a Boolean formula describing the set of initial states and the transition relation $T(\bar{x}, \bar{x}')$ is a Boolean formula describing the relation between one state and the next. A *state* in the system $\mathcal{S}$ is a Boolean assignment to the variables in $\bar{x}$. Abusing the notation, we denote the states in $\mathcal{S}$ by strings in $\Sigma^{|\bar{x}|}$. We will also view a formula $\varphi(\bar{x})$ as a set of states, i.e., the set of strings $\bar{a}$ where $\varphi[\bar{a}] = 1$.

A *trace* in $\mathcal{S}$ is a sequence of states $\bar{a}_0, \bar{a}_1, \bar{a}_2, \ldots$ such that $I[\bar{a}_0] = 1$ and for every $i \geqslant 0$, $T[\bar{a}_i, \bar{a}_{i+1}] = 1$. A state $\bar{a}$ is *reachable* (in $\mathcal{S}$), if there is a trace $\bar{a}_0, \bar{a}_1, \bar{a}_2, \ldots$ in which $\bar{a}$ appears. A formula $P(\bar{x})$ is *$\mathcal{S}$-invariant* if every state reachable in $\mathcal{S}$ satisfies P. It is *$\mathcal{S}$-inductive*, if $I(\bar{x}) \Rightarrow P(\bar{x})$ and $P(\bar{x}) \wedge T(\bar{x}, \bar{x}') \Rightarrow P(\bar{x}')$. Obviously, P is $\mathcal{S}$-invariant, whenever P is $\mathcal{S}$-inductive. The converse however is not true: It is possible that P is $\mathcal{S}$-invariant, but not $\mathcal{S}$-inductive, since the state that makes P not $\mathcal{S}$-inductive may actually be unreachable in $\mathcal{S}$. The formula P is often called a *safety property*.

Given a transition system $\mathcal{S}$ and property P, the PDR algorithm decides if P is $\mathcal{S}$-invariant. As output, it produces a sequence of Boolean formulas $F_0, F_1, \ldots, F_m$, all using variables in $\bar{x}$, such that:

- F_j is a formula that over-approximates the states that are reachable in at most j steps, for every $0 \leqslant j \leqslant m$.
- If P is $\mathcal{S}$-invariant, $F_m \Rightarrow P$ and F_m is $\mathcal{S}$-inductive.
- If P is not $\mathcal{S}$-invariant, it outputs a counterexample trace $\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_m$ constructed from $F_1, \ldots, F_m$ that violates the safety property P, i.e., $P[\bar{a}_m] = 0$.

For more details on the algorithm and its implementation, we refer interested readers to [23, 17, 18, 19, 24].

III. 2-DQBF Solving and Certification

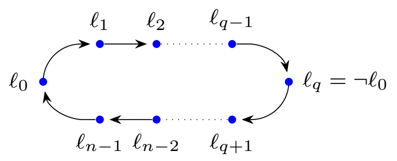
In this section we will show how to solve the satisfiability of 2-DQBF with PDR algorithm. We present a linear time reduction from 2-DQBFs to PDR instances in Section III-A. We then show how to extract the Skolem functions from the PDR certificates in Section III-B.

A. Linear time reduction from 2-DQBF to PDR

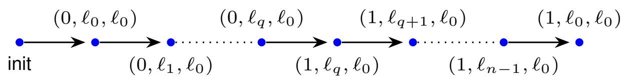
We first recall the approach for solving 2-SAT instances which inspires our use of PDR in solving 2-DQBF. Given a 2-CNF formula φ, we may assume w.l.o.g. that it is in implicative normal form, i.e., a conjunction of implications between literals: $\bigwedge_{1 \leqslant i \leqslant n}(\ell_{i,1} \to \ell_{i,2})$. It can be viewed as a directed graph $G = (V, E)$, called the implication graph of φ, where V is the set of all the literals in φ and $(\ell_{i,1}, \ell_{i,2})$ and $(\neg\ell_{i,2}, \neg\ell_{i,1})$ are edges in E for every $1 \leqslant i \leqslant n$. The formula φ is not satisfiable if and only if there is a contradicting cycle in G, i.e., a cycle that contains two contradicting literals. In other words, checking the (un)satisfiability of φ is equivalent to checking the existence of a contradicting cycle in G.

The main idea behind our encoding of 2-DQBF with a PDR instance is similar. The only difference is that in the 2-DQBF setting the edges in the implication graph are not explicitly given. Instead, they are succinctly represented by the matrix of the given 2-DQBF and it can be reduced to a PDR instance in which a counterexample trace corresponds to a contradicting cycle (in the implication graph of the expansion of 2-DQBF).

To illustrate, the following contradicting cycle:



where $\ell_q = \neg\ell_0$, can be encoded as a counterexample trace:



where init is the dummy state that serves as the initial state and the transition relation between two states can be constructed from the matrix of the 2-DQBF. The change of the first bit from 0 to 1 occurs when it reaches $(0, \ell_q, \ell_0)$ which indicates the existence of a path from ℓ_0 to $\neg\ell_0$. The safety property states that the system will not reach $(1, \ell, \ell)$ for any literal ℓ.

We now formally present the construction. Given a 2-DQBF $\Phi := \forall\bar{x}\exists y_0(\bar{z}_0)\exists y_1(\bar{z}_1)\ \phi(\bar{x}, y_0, y_1)$, where $|\bar{x}| = n$, we construct the transition system $\mathcal{S} = (\bar{r}, I, T)$ where each component is as follows.

The number of variables in $\bar{r}$ is $|\bar{r}| = 6+n+\max(|\bar{z}_0|, |\bar{z}_1|)$. For convenience, we denote $\bar{r}$ as the concatenation of the following vectors:

r_0	r_1	k	b	$\bar{x}$	k_T	b_T	$\bar{z}_T$

where $|r_0| = |r_1| = |k| = |b| = |k_T| = |b_T| = 1$, $|\bar{x}| = n$ and $|\bar{z}_T| = \max(|\bar{z}_0|, |\bar{z}_1|)$. Note that the variables in $\bar{x}$ are reused as variables in the transition system $\mathcal{S}$.

The intended meaning of $\bar{r}$ is to encode a tuple (a, ℓ_i, ℓ_0). The bit r_0 indicates whether it is the initial state. The bit r_1 is the "flag" bit indicating whether we have encountered $(1, \neg\ell_0, \ell_0)$. The vector $(k, b, \bar{x})$ corresponds to the literal $\ell_i := L^b_{k,\bar{z}_k}$, and $(k_T, b_T, \bar{z}_T)$ corresponds to the literal ℓ_0.

The initial condition $I(\bar{r})$ is $\bar{r} = 10\cdots0$ that encodes the dummy init state. The transition relation $T(\bar{r}, \bar{r}')$ is defined as $\varphi_1 \vee \varphi_2 \vee \varphi_3$ where φ_1 encodes the transition from the initial state, φ_2 encodes the transition from the state (r_1, ℓ_i, ℓ_0) to $(r_1', \ell_{i+1}, \ell_0)$, and φ_3 encodes the transition when the flag bit changes from 0 to 1. Formally, they are defined as follows.

$$\begin{aligned}
\varphi_1 \;&:= (\bar{r} = 10\cdots0) \wedge (r_0' = 0) \wedge (r_1' = 0) \\
&\quad \wedge \begin{cases} (k_T' = 0) \wedge ((b', \bar{z}_0') = (b_T', \bar{z}_T')) & \text{if } k' = 0 \\ (k_T' = 1) \wedge ((b', \bar{z}_1') = (b_T', \bar{z}_T')) & \text{if } k' = 1 \end{cases} \\
\varphi_2 \;&:= (r_0 = r_0' = 0) \wedge (r_1 = r_1') \wedge (k' = \neg k) \\
&\quad \wedge \bar{z}_0 \cap \bar{z}_1 = \bar{z}_0' \cap \bar{z}_1' \\
&\quad \wedge (k_T, b_T, \bar{z}_T) = (k_T', b_T', \bar{z}_T') \\
&\quad \wedge \begin{cases} \neg\phi[(\bar{x} \setminus \bar{z}_0)/(\bar{x}' \setminus \bar{z}_0'), y_0/b, y_1/\neg b'] & \text{if } k = 0 \\ \neg\phi[(\bar{x} \setminus \bar{z}_1)/(\bar{x}' \setminus \bar{z}_1'), y_0/\neg b', y_1/b] & \text{if } k = 1 \end{cases} \\
\varphi_3 \;&:= (r_0 = r_0' = 0) \wedge (r_1 = 0) \wedge (r_1' = 1) \\
&\quad \wedge (k, b, \bar{z}_k) = (k_T, \neg b_T, \bar{z}_T) \\
&\quad \wedge (k, b, \bar{z}_k) = (k', b', \bar{z}_k') \\
&\quad \wedge (k_T, b_T, \bar{z}_T) = (k_T', b_T', \bar{z}_T')
\end{aligned}$$

Remark 1. The formula φ_2 captures all the edges in the implication graph of $\exp(\Phi)$ in the sense that:

- For every $\bar{s}_0, \bar{s}_1 \in \Sigma^{|\bar{r}|}$, if $\varphi_2[\bar{s}_0, \bar{s}_1] = 1$, then $L^{d_0}_{a_0,\bar{c}_0} \vee L^{d_1}_{a_1,\bar{c}_1}$ is a clause in $\exp(\Phi)$ where $d_i = \bar{s}_i\big|_{\bar{r}\downarrow b}$, $a_i = \bar{s}_i\big|_{\bar{r}\downarrow k}$ and $\bar{c}_i = \bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_i}$ for $0 \leqslant i \leqslant 1$.
- Conversely, for every clause $L^{d_0}_{a_0,\bar{c}_0} \vee L^{d_1}_{a_1,\bar{c}_1}$ in $\exp(\Phi)$, there is $\bar{s}_0, \bar{s}_1 \in \Sigma^{|\bar{r}|}$, such that $\varphi_2[\bar{s}_0, \bar{s}_1] = 1$, where $d_i = \bar{s}_i\big|_{\bar{r}\downarrow b}$, $a_i = \bar{s}_i\big|_{\bar{r}\downarrow k}$ and $\bar{c}_i = \bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_i}$ for $0 \leqslant i \leqslant 1$.

Finally, the safety invariant property $P(\bar{r})$ is defined as:

$$P(\bar{r}) \;:=\; (k, b, \bar{z}_k) = (k_T, b_T, \bar{z}_T) \to \neg r_1$$

That $\mathcal{S}$ and P capture precisely the satisfiability of Φ is stated formally in Theorem 1.

Theorem 1. *Φ is not satisfiable if and only if the safety property P is not $\mathcal{S}$-invariant.*

Proof. We will show that there is a contradicting cycle in the implication graph of $\exp(\Phi)$ if and only if there is trace in $\mathcal{S}$ that violates the safety property $P(\bar{r})$.

(if) Suppose there is a counterexample trace $\bar{s}_0, \bar{s}_1, \ldots, \bar{s}_m$.

We will use the following notations. For every $1 \leqslant i \leqslant m$:

$$d_i = \bar{s}_i\big|_{\bar{r}\downarrow b},\ a_i = \bar{s}_i\big|_{\bar{r}\downarrow k} \text{ and } \bar{c}_i = \bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_i}}.$$

Also let:

$$d_T = \bar{s}_i\big|_{\bar{r}\downarrow b_T},\ a_T = \bar{s}_i\big|_{\bar{r}\downarrow k_T} \text{ and } \bar{c}_T = \bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_T}}.$$

Note that $\bar{s}_i\big|_{\bar{r}\downarrow b_T}$ and $\bar{s}_i\big|_{\bar{r}\downarrow k_T}$ stay the same for every $1 \leqslant i \leqslant m$, thus, d_T, a_T and $\bar{c}_T$ are well defined. Let L_i denote the literal $L^{d_i}_{a_i,\bar{c}_i}$, for each $1 \leqslant i \leqslant m$ and L_T the literal $L^{d_T}_{a_T,\bar{c}_T}$.

By the definition of $\mathcal{S}$ and P, we have:

$$\bar{s}_1\big|_{\bar{r}\downarrow r_1} = 0,\ \bar{s}_m\big|_{\bar{r}\downarrow r_1} = 1 \text{ and } L_1 = L_m = L_T.$$

Since $\bar{s}_1\big|_{\bar{r}\downarrow r_1}$ and $\bar{s}_m\big|_{\bar{r}\downarrow r_1}$ differ, there is an index $1 \leqslant q \leqslant m$ such that:

$$\bar{s}_q\big|_{\bar{r}\downarrow r_1} = 0 \text{ and } \bar{s}_{q+1}\big|_{\bar{r}\downarrow r_1} = 1.$$

That is, q is the index when the flag bit changes from 0 to 1. This change only happens when $\varphi_3[\bar{s}_q, \bar{s}_{q+1}] = 1$, which means L_q and L_T are contradicting literals.

Since φ_1 holds only on the dummy init state, for every $1 \leqslant i \leqslant m$ where $i \neq q$:

$$\varphi_2[(\bar{r}, \bar{r}')/(\bar{s}_i, \bar{s}_{i+1})] = 1.$$

By Remark 1, each $L_i \vee L_{i+1}$ is a clause in $\exp(\Phi)$, for every $1 \leqslant i \leqslant m-1$ where $i \neq q$. By routine inspection, these clauses form a cycle in the implication graph of $\exp(\Phi)$ that contains the literals L_T and L_q which are contradicting literals.

(only if) Suppose there is a contradicting cycle in the implication graph of $\exp(\Phi)$. Let the cycle be:

$$\ell_1 \to \cdots \to \ell_q \to \ell_{q+1} \to \cdots \to \ell_m = \ell_1.$$

where ℓ_q is $\neg\ell_1$.

For each $1 \leqslant i \leqslant m$, let $d_i, a_i, \bar{c}_i$ be such that $L^{d_i}_{a_i,\bar{c}_i} = \ell_i$. Let $d_T, a_T, \bar{c}_T$ be such that $L^{d_T}_{a_T,\bar{c}_T} = \ell_1$.

Consider the following trace $\bar{s}_0, \bar{s}_1, \ldots, \bar{s}_{m+1}$, where each $\bar{s}_i$ is as follows.

- $\bar{s}_0 = 10\cdots0$.
- For $1 \leqslant i \leqslant q$, $\bar{s}_i\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_i\big|_{\bar{r}\downarrow r_1} = 0$, $\bar{s}_i\big|_{\bar{r}\downarrow k} = a_i$, $\bar{s}_i\big|_{\bar{r}\downarrow b} = d_i$, and $\bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_i}} = \bar{c}_i$.
- For $i = q+1$, $\bar{s}_i\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_i\big|_{\bar{r}\downarrow r_1} = 1$, $\bar{s}_i\big|_{\bar{r}\downarrow k} = a_q$, $\bar{s}_i\big|_{\bar{r}\downarrow b} = \neg d_q$, and $\bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_q}} = \bar{c}_q$
- For $q+2 \leqslant i \leqslant m$, $\bar{s}_i\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_i\big|_{\bar{r}\downarrow r_1} = 1$, $\bar{s}_i\big|_{\bar{r}\downarrow k} = a_{i-1}$, $\bar{s}_i\big|_{\bar{r}\downarrow b} = d_{i-1}$, and $\bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_{i-1}}} = \bar{c}_{i-1}$.
- For $1 \leqslant i \leqslant m+1$, $\bar{s}_i\big|_{\bar{r}\downarrow k_T} = a_T$, $\bar{s}_i\big|_{\bar{r}\downarrow b_T} = d_T$ and $\bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_T}} = \bar{c}_T$.

By routine inspection, $\bar{s}_0, \ldots, \bar{s}_{m+1}$ is a counterexample trace that violates the safety property P. □

B. *Proof extraction and model extraction*

In this section, we will show how to extract the certificate for the original 2-DQBF from the certificate provided by the PDR algorithm. Let Φ be the given 2-DQBF as in the previous section and let $\mathcal{S} = (\bar{r}, I, T)$ and P be the constructed transition system and the safety property.

If Φ is unsatisfiable, the PDR algorithm would give us a counterexample trace $\bar{s}_0, \ldots, \bar{s}_m$ from which we can construct the contradicting cycle in the implication graph of $\exp(\Phi)$ as described in the (if) direction in the proof of Theorem 1. Such a contradicting cycle is the certificate for the unsatisfiability.

Now, consider the case when Φ is satisfiable. By Theorem 1, P is $\mathcal{S}$-invariant and the PDR algorithm outputs a Boolean formula $S(\bar{r})$ that is $\mathcal{S}$-invariant and also is an over-approximation of the reachable states from the initial states. We will show how to extract the Skolem functions f_0 and f_1 for y_0 and y_1, respectively, from the formula $S(\bar{r})$.

We first describe the main idea. Let G_Φ be the implication graph of $\exp(\Phi)$. Let $\mathrm{Tr}(G_\Phi)$ be the transitive closure of G_Φ. To avoid clutter with parentheses, we write $L \to L'$ to denote an edge from L to L'. The graph $\mathrm{Tr}(G_\Phi)$ will serve as the guide in constructing the Skolem functions for Φ. The intuition is that if the edge $X_{i,\bar{c}} \to \neg X_{i,\bar{c}}$ is present in $\mathrm{Tr}(G_\Phi)$, then we have to assign $X_{i,\bar{c}}$ to 0, which corresponds to the function f_i where $f_i(\bar{c}) = 0$. Similarly, if the edge $\neg X_{i,\bar{c}} \to X_{i,\bar{c}}$ is present in $\mathrm{Tr}(G_\Phi)$, then we have to assign $X_{i,\bar{c}}$ to 1, which corresponds to the function f_i where $f_i(\bar{c}) = 1$. If both edges are not present in $\mathrm{Tr}(G_\Phi)$, we can freely assign $X_{i,\bar{c}}$ to either 0 or 1. This intuition motivates us to introduce the following definition.

Definition 1. Let $0 \leqslant i \leqslant 1$ and $\bar{c} \in \Sigma^{|\bar{z}_i|}$. The variable $X_{i,\bar{c}}$ is *free* (w.r.t. Φ), if both edges $X_{i,\bar{c}} \to \neg X_{i,\bar{c}}$ and $\neg X_{i,\bar{c}} \to X_{i,\bar{c}}$ are not in $\mathrm{Tr}(G_\Phi)$.

We make a few observations stated formally below that give us the criterion the satisfying Skolem functions should obey.

(O1) If a variable $X_{i,\bar{c}}$ is free, there are Skolem functions (f_0, f_1) where $f_i(\bar{c}) = 0$ and (g_0, g_1) where $g_i(\bar{c}) = 1$. In other words, if $X_{i,\bar{c}}$ is free, we can assign the value of $f_i(\bar{c})$ to either 0 or 1.

(O2) If $X_{i,\bar{c}} \to \neg X_{i,\bar{c}}$ is an edge in $\mathrm{Tr}(G_\Phi)$, then the value of $f_i(\bar{c})$ must be 0 for every Skolem function f_0, f_1 for Φ.

(O3) If $\neg X_{i,\bar{c}} \to X_{i,\bar{c}}$ is an edge in $\mathrm{Tr}(G_\Phi)$, then the value of $f_i(\bar{c})$ must be 1.

(O4) It is not possible that both $X_{i,\bar{c}} \to \neg X_{i,\bar{c}}$ and $\neg X_{i,\bar{c}} \to X_{i,\bar{c}}$ are edges in $\mathrm{Tr}(G_\Phi)$, since both edges forms a contradicting cycle, which will contradict the assumption that Φ is satisfiable.

The main technical difficulty in constructing the Skolem functions is that we do not have the graph $\mathrm{Tr}(G_\Phi)$ explicitly, but only the formula $S(\bar{r})$. To connect $S(\bar{r})$ with $\mathrm{Tr}(G_\Phi)$, we view the formula $S(\bar{r})$ as a graph G_S, where the set of vertices is the same as the set of vertices in G_Φ and the set of edges is as follows. For $b_1, k_1, b_2, k_2 \in \Sigma$, for $\bar{c}_1 \in \Sigma^{|\bar{z}_{k_1}|}$ and $\bar{c}_2 \in \Sigma^{|\bar{z}_{k_2}|}$, $(L^{b_1}_{k_1,\bar{c}_1}, L^{b_2}_{k_2,\bar{c}_2})$ is an edge in G_S if and only if

$$S[0, 0, k_2, b_2, \mathrm{Ext}_{\bar{x}}(\bar{c}_2, \bar{z}_{k_2}), k_1, b_1, \bar{c}_1] = 1, \tag{2}$$

where $\mathrm{Ext}_{\bar{x}}(\bar{c}_2, \bar{z}_{k_2})$ is the assignment of $\bar{x}$ where all variables in $\bar{z}_{k_2}$ are assigned according to $\bar{c}$ and all variables in $\bar{x} \setminus \bar{z}_{k_2}$ are assigned with 0.

The intuition of Eq. (2) is as follows. Recall that the states in $\mathcal{S}$ represent the tuple (b, ℓ_i, ℓ_0) for some literals ℓ_i, ℓ_0. The set of reachable states in $\mathcal{S}$ are such tuples where there is a path from ℓ_0 to ℓ_i in the graph G_Φ, or equivalently, $\ell_0 \to \ell_i$ is an edge in $\mathrm{Tr}(G_\Phi)$. Now the graph G_S can be viewed as an over-approximation of $\mathrm{Tr}(G_\Phi)$, i.e., it contains all the edges in $\mathrm{Tr}(G_\Phi)$ and possibly some other edges that are not in $\mathrm{Tr}(G_\Phi)$.

Lemma 1 below states some useful facts on G_S.

Lemma 1.
- *The set of edges in G_S is an over-approximation of the set of edges in Tr(G_Φ).*
- *If $L_1 \to L_2$ is an edge in G_S and $L_2 \to L_3$ is an edge in Tr(G_Φ), then $L_1 \to L_3$ is an edge in G_S.*
- *If $L \to \neg L$ is an edge in Tr(G_Φ), then $\neg L \to L$ is not an edge in G_S.*

Proof. For the first bullet item, let $L \to L'$ be an edge in $\mathrm{Tr}(G_\Phi)$. Since $\mathrm{Tr}(G_\Phi)$ is the transitive closure of G_Φ, there is a path from L to L' in G_Φ, say:

$$L = L_1 \to L_2 \to \cdots \to L_m = L'$$

Let $d_i, a_i, \bar{c}_i$ be such that $L^{d_i}_{a_i,\bar{c}_i} = L_i$. Then in the transition system $\mathcal{S} = (\bar{r}, I, T)$, consider the states:

$$\bar{s}_0, \bar{s}_1, \ldots, \bar{s}_m,$$

where $\bar{s}_0 = 10 \cdots 0$ and for each $1 \leqslant i \leqslant m$:

- $\bar{s}_i\big|_{\bar{r}\downarrow r_0} = 0$,
- $\bar{s}_i\big|_{\bar{r}\downarrow r_1} = 0$,
- $\bar{s}_i\big|_{\bar{r}\downarrow k} = a_i$,
- $\bar{s}_i\big|_{\bar{r}\downarrow b} = d_i$,
- $\bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_i}} = \bar{c}_i$,
- $\bar{s}_i\big|_{\bar{r}\downarrow k_T} = a_1$,
- $\bar{s}_i\big|_{\bar{r}\downarrow b_T} = d_1$,
- $\bar{s}_i\big|_{\bar{r}\downarrow \bar{z}_{a_T}} = \bar{c}_1$.

It is routine to verify that $\bar{s}_0, \bar{s}_1, \ldots, \bar{s}_m$ is a trace in $\mathcal{S}$. In particular, the state $\bar{s}_m$ is reachable, i.e.:

$$S[0, 0, a_m, d_m, \mathrm{Ext}_{\bar{x}}(\bar{c}_m), a_1, d_1, \bar{c}_1] = 1.$$

Hence by the definition of G_S, $L_0 \to L_n$ is an edge in G_S, and thus, $\mathrm{Tr}(G_\Phi)$ is a subgraph of G_S.

For the second bullet item, let $L_i = L^{d_i}_{a_i,\bar{c}_i}$ for $i = 1, 2, 3$ with $L_1 \to L_2$ being an edge in G_S and $L_2 \to L_3$ being an edge in $\mathrm{Tr}(G_\Phi)$. Consider the states $\bar{s}_0, \bar{s}_1, \bar{s}_2, \bar{s}_3$, where $\bar{s}_0 = 10 \cdots 0$ and for each $1 \leqslant i \leqslant 3$:

- $\bar{s}_i\big|_{\bar{r}\downarrow r_0} = 0$,
- $\bar{s}_i\big|_{\bar{r}\downarrow r_1} = 0$,
- $\bar{s}_i\big|_{\bar{r}\downarrow k} = a_i$,
- $\bar{s}_i\big|_{\bar{r}\downarrow b} = d_i$,

- $\bar{s}_i\big|_{\bar{r}\downarrow\bar{z}_{a_i}} = \bar{c}_i$,
- $\bar{s}_i\big|_{\bar{r}\downarrow k_T} = a_1$,
- $\bar{s}_i\big|_{\bar{r}\downarrow b_T} = d_1$, and
- $\bar{s}_i\big|_{\bar{r}\downarrow\bar{z}_{a_T}} = \bar{c}_1$.

By similar arguments as the first bullet item, we can verify that the state $\bar{s}_1$ is reachable from $\bar{s}_0$, $\bar{s}_2$ is reachable from $\bar{s}_1$, and $\bar{s}_3$ is reachable from $\bar{s}_2$. Hence,

$$S[0, 0, a_3, d_3, \mathrm{Ext}_{\bar{x}}(\bar{c}_3), a_1, d_1, \bar{c}_1] = 1,$$

and $L_1 \to L_3$ is an edge in G_S.

For the third bullet item, let $L \to \neg L$ is an edge in $\mathrm{Tr}(G_\Phi)$ and let $L = L^d_{a,\bar{c}}$. Suppose to the contrary that $\neg L \to L$ is an edge in G_S. Consider the states $\bar{s}_0, \bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4$, where:

- $\bar{s}_0 = 10\cdots 0$
- $\bar{s}_1\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_1\big|_{\bar{r}\downarrow r_1} = 0$, $\bar{s}_1\big|_{\bar{r}\downarrow k} = a$, $\bar{s}_1\big|_{\bar{r}\downarrow b} = \neg d$, $\bar{s}_1\big|_{\bar{r}\downarrow\bar{z}_{a_i}} = \bar{c}$,
- $\bar{s}_2\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_2\big|_{\bar{r}\downarrow r_1} = 0$, $\bar{s}_2\big|_{\bar{r}\downarrow k} = a$, $\bar{s}_2\big|_{\bar{r}\downarrow b} = d$, $\bar{s}_2\big|_{\bar{r}\downarrow\bar{z}_{a_i}} = \bar{c}$,
- $\bar{s}_3\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_3\big|_{\bar{r}\downarrow r_1} = 1$, $\bar{s}_3\big|_{\bar{r}\downarrow k} = a$, $\bar{s}_3\big|_{\bar{r}\downarrow b} = d$, $\bar{s}_3\big|_{\bar{r}\downarrow\bar{z}_{a_i}} = \bar{c}$,
- $\bar{s}_4\big|_{\bar{r}\downarrow r_0} = 0$, $\bar{s}_4\big|_{\bar{r}\downarrow r_1} = 1$, $\bar{s}_4\big|_{\bar{r}\downarrow k} = a$, $\bar{s}_4\big|_{\bar{r}\downarrow b} = d$, $\bar{s}_4\big|_{\bar{r}\downarrow\bar{z}_{a_i}} = \bar{c}$,
- For $1 \leqslant i \leqslant 4$, $\bar{s}_i\big|_{\bar{r}\downarrow k_T} = a$, $\bar{s}_i\big|_{\bar{r}\downarrow b_T} = \neg d$, and $\bar{s}_i\big|_{\bar{r}\downarrow\bar{z}_{a_T}} = \bar{c}$.

It is routine to check that $\bar{s}_i$ is reachable from $\bar{s}_{i-1}$ for every $1 \leqslant i \leqslant 4$. In particular, the state $\bar{s}_4$ violates the property P, which is a contradiction to the existence of S. Hence $\neg L \to L$ is not an edge in G_S. ☐

Now consider the following candidate Skolem functions for $0 \leqslant i \leqslant 1$:

$$\begin{aligned} f_i(\bar{z}_i) \;:=\; & S[0, 0, i, 1, \mathrm{Ext}_{\bar{x}}(\bar{z}_i), i, 0, \bar{z}_i] = 1 \;\wedge \\ & S[0, 0, i, 0, \mathrm{Ext}_{\bar{x}}(\bar{z}_i), i, 1, \bar{z}_i] = 0 \end{aligned}$$

where $\mathrm{Ext}_{\bar{x}}(\bar{z}_i)$ denotes the substitution of all variables in $\bar{x} \setminus \bar{z}_i$ with 0. Intuitively it means that for every $\bar{c} \in \Sigma^{|\bar{z}_i|}$, we assign $f_i(\bar{c}) = 1$ if and only if $L^0_{i,\bar{c}} \to L^1_{i,\bar{c}}$ is an edge in G_S and $L^1_{i,\bar{c}} \to L^0_{i,\bar{c}}$ is not an edge in G_S.

Note that after the first call of the PDR algorithm, the candidate Skolem functions are not necessary the correct ones, because the formula $S(\bar{r})$ is only an over-approximation of the reachable states, as shown in the following example.

Example 1. Let $\psi := \forall x\ \exists y_0(x) \exists y_1(x)\ y_0 \neq y_1$, which is obviously satisfiable. The first call of the PDR algorithm gives us the formula $S(\bar{r})$ where the graph G_S is depicted in Figure 2. The candidate Skolem functions f_0, f_1 defined by G_S are the constant function 0, which obviously are not the correct Skolem functions for ψ.

To verify that the candidate functions f_0, f_1 are indeed the Skolem functions for Φ, we check the satisfiability of the formula:

$$\neg\phi(\bar{x}, y_0, y_1) \;\wedge\; y_0 = f_0(\bar{z}_0) \;\wedge\; y_1 = f_1(\bar{z}_1) \qquad (3)$$

$$\begin{tikzcd}
X_{1,0} \arrow[dr, shift left] & \neg X_{1,0} \arrow[dl, shift left] & X_{1,1} \arrow[dr, shift left] & \neg X_{1,1} \arrow[dl, shift left] \\
X_{0,0} \arrow[u] \arrow[ur, shift left] \arrow[r, shift left] & \neg X_{0,0} \arrow[u] \arrow[ul, shift left] \arrow[l, shift left] & X_{0,1} \arrow[u] \arrow[ur, shift left] \arrow[r, shift left] & \neg X_{0,1} \arrow[u] \arrow[ul, shift left] \arrow[l, shift left]
\end{tikzcd}$$

Fig. 2: The transitive closure of the implication graph of the expansion $\exp(\psi)$ has only the black edges, while the graph G_S also contains the red edges.

If it is unsatisfiable, then f_0, f_1 are indeed Skolem functions for y_0, y_1. Otherwise, we need to refine either f_0 or f_1. Let $(\bar{a}, b_0, b_1)$ be a satisfying assignment of the formula in Eq. (3). Let $\bar{c}_0 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_0}$ and $\bar{c}_1 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_1}$. It implies the clause $C_{\bar{a},b_0,b_1}$ in $\exp(\Phi)$ is violated, i.e., both literals $L^{\neg b_0}_{0,\bar{c}_0}$ and $L^{\neg b_1}_{1,\bar{c}_1}$ have value 0. Note also that since $\phi(\bar{a}, b_0, b_1) = 0$, by definition, the clause $C_{\bar{a},b_0,b_1}$ is in $\exp(\Phi)$, hence, both $L^{b_0}_{0,\bar{c}_0} \to L^{\neg b_1}_{1,\bar{c}_1}$ and $L^{b_1}_{1,\bar{c}_1} \to L^{\neg b_0}_{0,\bar{c}_0}$ are edges in G_Φ, hence, in $\mathrm{Tr}(G_\Phi)$. In this case we can show that both $X_{0,\bar{c}_0}$ and $X_{1,\bar{c}_1}$ are free, as stated formally in the following lemma.

Lemma 2. *Suppose Φ is satisfiable and suppose $(\bar{a}, b_0, b_1)$ is a satisfying assignment of the formula in Eq.* (3). *Let $\bar{c}_0 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_0}$ and $\bar{c}_1 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_1}$. Then, both $X_{0,\bar{c}_0}$ and $X_{1,\bar{c}_1}$ are free.*

Proof. Since $(\bar{a}, b_0, b_1)$ is a satisfying assignment of the formula in Eq. (3), it is also a satisfying assignment for $\neg\phi$. Thus, the clause $C_{\bar{a},b_0,b_1}$ is in $\exp(\Phi)$, which means that:

$$L^{b_1}_{1,\bar{c}_1} \to L^{\neg b_0}_{0,\bar{c}_0} \text{ and } L^{b_0}_{0,\bar{c}_0} \to L^{\neg b_1}_{1,\bar{c}_1}$$

are both edges in G_Φ, hence, in $\mathrm{Tr}(G_\Phi)$.

Assume to the contrary that at least one of $X_{0,\bar{c}_0}$ and $X_{1,\bar{c}_1}$ is not free. We first assume that $X_{0,\bar{c}_0}$ is not free, i.e., one of $L^{\neg b_0}_{0,\bar{c}_0} \to L^{b_0}_{0,\bar{c}_0}$ and $L^{b_0}_{0,\bar{c}_0} \to L^{\neg b_0}_{0,\bar{c}_0}$ is an edge in $\mathrm{Tr}(G_\Phi)$. The case when $X_{1,\bar{c}_1}$ is not free can be treated in a similar manner.

If $L^{\neg b_0}_{0,\bar{c}_0} \to L^{b_0}_{0,\bar{c}_0}$ is an edge $\mathrm{Tr}(G_\Phi)$, then there is a sequence of edges in $\mathrm{Tr}(G_\Phi)$:

$$L^{b_1}_{1,\bar{c}_1} \to L^{\neg b_0}_{0,\bar{c}_0} \to L^{b_0}_{0,\bar{c}_0} \to L^{\neg b_1}_{1,\bar{c}_1}.$$

Since $\mathrm{Tr}(G_\Phi)$ is the transitive closure of G_ϕ, the edge $L^{b_1}_{1,\bar{c}_1} \to L^{\neg b_1}_{1,\bar{c}_1}$ is also in $\mathrm{Tr}(G_\Phi)$ and hence in G_S. By Lemma 1, $L^{\neg b_1}_{1,\bar{c}_1} \to L^{b_1}_{1,\bar{c}_1}$ is not an edge in G_S. By the construction of f_1, we have $f_1(\bar{c}_1) = \neg b_1$, which contradicts the assumption that $(\bar{a}, b_0, b_1)$ is a satisfying assignment of $\neg\phi \wedge y_0 = f_0 \wedge y_1 = f_1$.

If $L^{b_0}_{0,\bar{c}_0} \to L^{\neg b_0}_{0,\bar{c}_0}$ is an edge in $\mathrm{Tr}(G_\Phi)$, by Lemma 1, $L^{b_0}_{0,\bar{c}_0} \to L^{\neg b_0}_{0,\bar{c}_0}$ is an edge in G_S and $L^{\neg b_0}_{0,\bar{c}_0} \to L^{b_0}_{0,\bar{c}_0}$ is not an edge in G_S. By the construction of f_0, we have $f_0(\bar{c}_0) = \neg b_0$, which contradicts the assumption that $(\bar{a}, b_0, b_1)$ is a satisfying assignment of $\neg\phi \wedge (y_0 = f_0) \wedge (y_1 = f_1)$. ☐

Remark 2. It is worth noting that Lemma 2 does not contradict with the fact that (f_0, f_1) are not the correct Skolem functions for Φ. It is possible that there are correct Skolem functions

$$\begin{tikzcd}
X_{1,0} \arrow[dr, shift left] & \neg X_{1,0} \arrow[l] \arrow[d] \arrow[dl, shift left] & & X_{1,1} \arrow[dr, shift left] & \neg X_{1,1} \arrow[l] \arrow[d] \arrow[dl, shift left] \\
X_{0,0} \arrow[u] \arrow[r] \arrow[ur, shift left] & \neg X_{0,0} \arrow[ul, shift left] & & X_{0,1} \arrow[u] \arrow[r] \arrow[ur, shift left] & \neg X_{0,1} \arrow[ul, shift left]
\end{tikzcd}$$

Fig. 3: The closure of the original implication graph are the black edges in black. Edges added to the closure after adding $X_{0,0} \rightarrow \neg X_{0,0}$ are in blue. The red edges are the over-aprroximated edges.

g_0, g_1, where g_0 agrees with f_0 on $\bar{c}_0$, but differ from f_1 on $\bar{c}_1$, or that g_0 differs from f_0 on $\bar{c}_0$, but agrees with f_1 on $\bar{c}_1$.

To "fix" the candidate functions f_0, f_1, we add edges into the graph $G_{\mathcal{S}}$. Consider the 2-DQBF ψ in Example 1. Taking the constant functions $f_0, f_1 = 0$, the formula in Eq. (3) has a satisfying assignment: $x = 0, y_0 = 0, y_1 = 0$. Lemma 2 implies that both $X_{0,0}$ and $X_{1,0}$ are free. We can "refine" the function f_i by adding the edge $X_{0,0} \rightarrow \neg X_{0,0}$, which is equivalent to fixing the value $f_0(x)$ to 0. Note that to add the edge $X_{0,0} \rightarrow \neg X_{0,0}$ into the graph $G_{\mathcal{S}}$, we only need to add it to the transition of the system $\mathcal{S}$. After calling the PDR algorithm again, the graph $G_{\mathcal{S}}$ is now as in Figure 3 and the candidate Skolem functions become $f_0 = 0, f_1 = 1$, which are indeed correct Skolem functions of ψ.

We formalise this idea in Algorithm 1. The while-loop corresponds to the refinement of the candidate Skolem functions. In Line 10 we force an assignment the assignment $f_{0,\bar{c}_0} = b_0$ by adding the edge $L_{0,\bar{c}_0}^{\neg b_0} \rightarrow L_{0,\bar{c}_0}^{b_0}$ to the transition relation T. The correctness of Algorithm 1 is stated formally in Theorem 2.

Theorem 2. *Algorithm 1 is correct.*

Proof. The correctness of the reduction to the PDR instance $\mathcal{S}$ and P follows from Theorem 1. What is left is to show that the output Skolem functions f_0, f_1 are indeed the satisfying Skolem functions for Φ (when Φ is satisfiable).

It suffices to show that the refinement step is correct. Suppose Φ is satisfiable. Let $(\bar{a}, b_0, b_1)$ be a satisfying assignment of $\neg\varphi \wedge y_0 = f_0 \wedge y_1 = f_1$. Lemma 2 implies that both $f_0(\bar{c}_0)$ and $f_1(\bar{c}_1)$ are free, where $\bar{c}_0 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_0}$ and $\bar{c}_1 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_1}$. Thus, there are satisfying Skolem functions for Φ regardless of what we choose to assign in the refinement step.

In each refinement step, we force one assignment, and the number of free variables decreases by at least one per refinement. Hence the algorithm will terminate and the output (f_0, f_1) are correct Skolem functions. □

Note that in the refinement step in Algorithm 1, we choose to force $f_0(\bar{c}_0) = b_0$, which will implicitly force $f_1(\bar{c}_1) = \neg b_1$, due to the clause $C_{\bar{a},b_0,b_1}$ in $\exp(\Phi)$. Alternatively, we may choose to force $f_0(\bar{c}_0) = b_0$, but this choice will not implicitly force the value of $f(\bar{c}_1)$. In our experiments we implement the

Algorithm 1 2DQR

```
    Input: 2-DQBF $\Phi := \forall\bar{x}\ \exists y_0(\bar{z}_0)\exists y_1(\bar{z}_1)\ \varphi(\bar{x}, y_0, y_1)$
 1: Run the reduction as in Section III-A on $\Phi$.
 2: Let $\mathcal{S} = (\bar{r}, I, T)$ and $P$ be the output.
 3: Run the PDR algorithm on $\mathcal{S}$ with the safety property $P$
 4: if $P$ is $\mathcal{S}$-invariant then                    ▷ The input $\Phi$ is satisfiable
 5:     Let $S(\bar{r})$ be the inductive safe set formula.
 6:     Construct the formulas $f_0(\bar{z}_0)$ and $f_1(\bar{z}_1)$ based on $S$.
 7:     while $\neg\varphi \wedge y_0 = f_0 \wedge y_1 = f_1$ is satisfiable do      ▷ Refinement
 8:         Let $(\bar{a}, b_0, b_1)$ be the satisfying assignment.
 9:         Let $\bar{c}_0 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_0}$ and $\bar{c}_1 = \bar{a}\big|_{\bar{x}\downarrow\bar{z}_1}$.
10:         $T \leftarrow T \vee \textsc{Assign}(0, \bar{c}_0, b_0)$           ▷ Forcing an assignment
11:         Call the PDR algorithm on $\mathcal{S} = (\bar{r}, I, T)$ and $P$.
12:         Let $S(\bar{r})$ be the inductive safe set formula.
13:         Construct the formulas $f_0(\bar{z}_0)$ and $f_1(\bar{z}_1)$ based on $S$.
14:     return $f_0, f_1$.
15: else                                              ▷ The input $\Phi$ is not satisfiable
16:     return UNSAT.

17: procedure $\textsc{Assign}(i, \bar{c}, b_0)$
18:     return $(r_0 = r_0' = 1) \wedge (r_1 = r_1') \wedge (k = k' = i) \wedge (\bar{z}_i = \bar{z}_i' = \bar{c}) \wedge (\neg b = b' = b_0) \wedge (k_T, b_T, \bar{z}_T) = (k_T', b_T', \bar{z}_T')$
```

forcing of $f_0(\bar{c}_0) = b_0$, which we believe is more efficient than the other due to the "implicit" forcing.

IV. Experimental Evaluation

Benchmarks: We generate two families of benchmarks: PEC and succinct graph 2-colorability, which are then converted to 2-DQBF.

(PEC) We generate PEC instances with two black-boxes from the ISCAS89 benchmarks [25], where we randomly choose two "sub-circuits" from each circuit and replace them with two black-boxes. We ensure that the dependency sets of the sub-circuits ranges from "being disjoint" to "being almost equal". The instances have 33-674 (universal) variables and the states in the constructed transition systems have 61-1071 bits. In total, there are 624 instances. Each instance is post-processed with the command `fraig` in `ABC` [26] and the resulting test case is in circuit form.

The above method would generate satisfiable instances. We obtain unsatisfiable instances by swapping the dependencies set. Though such swapping does not always guarantee unsatisfiability, but it almost always produces unsatisfiable instances.

(Succinct graph 2-colorability) This family of benchmarks is based on the succinct graph models introduced in [27] where, instead of being given the list of edges in the graph, we are given a Boolean circuit that represents the edges in the graph. Let $C(\bar{u}, \bar{v})$ be a Boolean circuit where $|\bar{u}| = |\bar{v}| = n$. It represents a graph G_C where $\{0,1\}^n$ is the set of vertices and two vertices $\bar{a}$ and $\bar{b}$ are adjacent if and only if $C(\bar{a}, \bar{b}) = 1$. The problem of *succinct graph 2-colorability* is defined as: Given a circuit C, decide if the graph G_C is 2-colorable.

We generate 2-colorability instances by first generating two random permutation circuits $D, D' : \{0,1\}^n \rightarrow \{0,1\}^n$. We consider the graph where (x, x') is an edge if the first bit of $D(x)$ is the same as the first bit of $D'(x')$. If we let $D' = D$, the graph defined by C is bipartite, i.e. 2-colorable. Otherwise, the graph is unlikely to be bipartite.

Each random permutation is constructed in m rounds. In each round, we randomly pick $k \in [n]$, generate a clause $c \subseteq \{x_1, \cdots, x_n\}$ with $\Pr[x_i \in c] = \frac{1}{2}$ for $i \neq k$ and $x_k \notin c$, and let x_k to be $x_k \oplus c$. We generate one instance for each $n \in \{2, \cdots, 127\}$ and set $m = 2n$. Again, each instance is post-processed with the command `fraig` in `ABC` [26]. The resulting instances are in circuit form.

We then use the following reduction to obtain 2-DQBF. On input circuit $C(\bar{u}, \bar{v})$ where $|\bar{u}| = |\bar{v}| = n$, let Φ be the following 2-DQBF:

$$\forall \bar{x}_1 \forall \bar{x}_2\ \exists y_1(\bar{x}_1) \exists y_2(\bar{x}_2) \quad \left(\bar{x}_1 = \bar{x}_2\ \rightarrow\ y_1 = y_2\right) \\ \wedge\ \left(C(\bar{x}_1, \bar{x}_2)\ \rightarrow\ y_1 \neq y_2\right)$$

where $|\bar{x}_1| = |\bar{x}_2| = n$. Intuitively, we regard the values $0, 1$ as the colors and view a coloring on the vertices as a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. The formula Φ states that y_1 and y_2 must represent the same function and that two adjacent vertices have different colors. Thus, Ψ is satisfiable if and only if the graph G_C is 2-colorable.

Setup: We implement our method, which we call `2DQR`, using `AVR` [19] as the PDR solver and `Z3` [28] as the SAT solver and a parser for the SMT-LIB 2 format. We compare its performance (with or without Skolem function generation for satisfiable test cases) with `DQBDD` [13], `HQS` [10] and `Pedant` [11, 12]. We run `DQBDD` without generating the Skolem functions, while `HQS` and `Pedant` are run both with and without Skolem function generation. The latest version of `HQS` does not support Skolem function generation. When the Skolem functions are needed, we use an older version of `HQS`. Otherwise, we use the latest version. Since `HQS` and `Pedant` do not take circuit form as input, Tseitin transformation is applied to the instances before being fed to `HQS` and `Pedant`. Each solver had 600 seconds to solve each instance. We run the experiments on Ubuntu 22.04.3 LTS with 48 GB of 2400 MHz DDR4 memory and i5-13400 CPU.

Results: In the first batch of experiments, we compare `2DQR` with other DQBF solvers on the PEC instances. The cactus plots in Figure 4a show the results, where the horizontal axis corresponds to the running time (s) and the vertical axis to the number of solved instances. `2DQR` means without Skolem function generation and `2DQR_skolem` means with Skolem function generation. The time is measured starting from when the input DQBF is read until it terminates/time out, i.e., it includes *the reduction time to PDR instance*, *the pre-processing step `fraig` in `ABC`* and *the output generation*.

For satisfiable instances, `2DQR` outperforms the other solvers by large margins. We remark that the Skolem function generation introduces little run-time overhead since in most cases, they need very few extra calls to the PDR solver. For unsatisfiable instances, `2DQR` outperforms `HQBDD` and `HQS`, while `Pedant` outperformed `2DQR` by a small margin.

Next, we provide a pairwise comparison between our method and other solvers. The scatter plots in the log scale are shown in Figure 4b, where each point represents an instance. The horizontal axis corresponds to the time spent by `2DQR` and the vertical axis represents that by the compared solver.

In most plots, there are a lot of points lying on the bottom right plane, a lot of which have minor differences and are solved within 10 seconds by both methods. Also, there are a lot of points lying on the top boundary of the graph, indicating that there are a lot of cases that are solved by `2DQR` but not the others. As for the graph of `Pedant` v.s. `2DQR` on the unsatisfiable cases, the dots are quite close to the center line, and there are only four cases in which `Pedant` solves but `2DQR` does not.

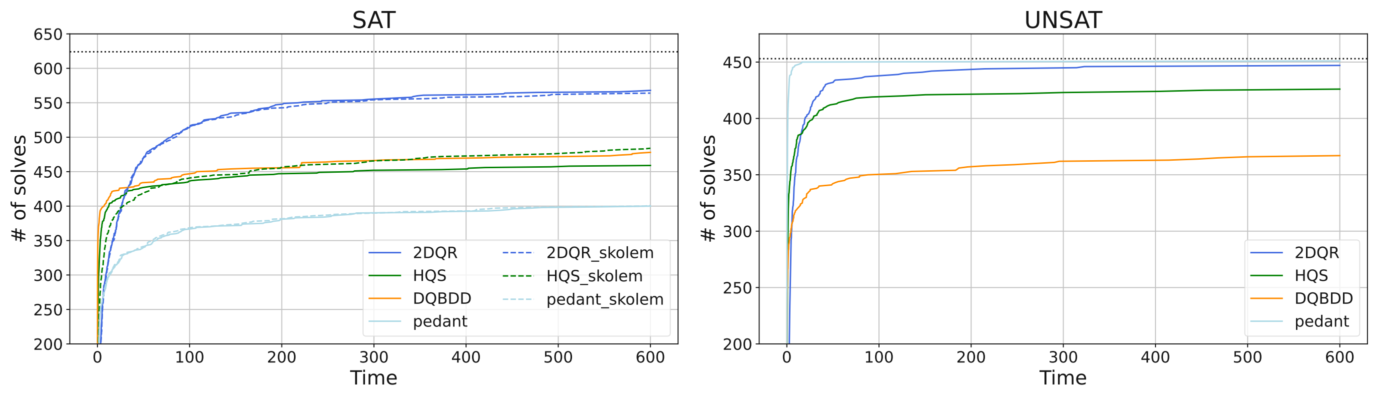
Next, we analyze the circuit size generated by the three methods, `2DQR` generates an AIG in SMT2 format, while `HQS` and `Pedant` generate an AIG in AIGER format. For a fair comparison, we first remove the Tseitin variables from the Skolem function given by `HQS` and `Pedant` using `ABC`, which is done by first removing the definition from the file and running `read skolem.aig; write skolem.aig` on `ABC`. As for `2DQR`, we transform the SMT2 format to verilog format and run `read skolem.v; strash; write skolem.aig`. We also use `fraig` in `ABC` to do some optimization.

For the post-processing steps above, `2DQR` takes 26 seconds, `2DQR` with `fraig` 33 seconds, `HQS` 10 seconds, `HQS` with `fraig` 28 seconds, and `Pedant` 74 seconds. However, `Pedant` with `fraig` takes more than a day to post-process, so we exclude it here. In Figure 4c, we plot the number of AND gates in the Skolem function as a scatter plot to give a pairwise comparison between solvers. Only instances solved by both solvers appear in this plot. It shows that `2DQR` and `HQS` performed similarly, and both outperformed `Pedant` with quite a large margin. When `fraig` is used, that are slight reductions on the number of AND gates for every solver. Figure 4d shows that the performance of `2DQR` and `HQS` are similar, but `2DQR` and `2DQR` with `fraig` are slightly better than `HQS` and `HQS` with `fraig`. Note also that `Pedant`'s distribution is not close to the others.

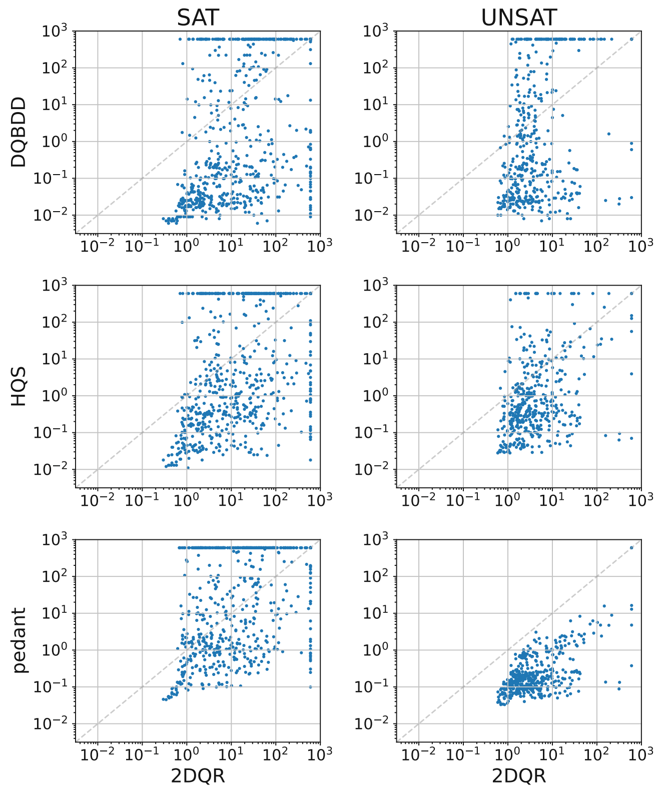
In the construction of Skolem functions, on most PEC instances, there is no extra call to the PDR algorithm. Out of 624 instances, two require 1 extra call, one requires 2 extra calls, one requires 17 extra calls, one requires 22 extra calls and one 29 extra calls.

In the second batch of experiments, we consider the 2-colorability instances. Figure 5 shows the results, where the horizontal axis corresponds to the number of bits of the graph, and the vertical axis corresponds to the time needed to solve the instance. The vertical axis is in log scale for better resolution. For the satisfiable instances, `2DQR` outperforms the other solvers by quite a large margin. Here, each instance needs at least one more extra PDR call for the generation of the Skolem function, which is expected as we need to choose a color to assign to a partition before we can get a coloring. For the unsatisfiable instances, `2DQR`, `DQBDD`, and `HQS` could not solve any instance of size at least 12 bits, but `Pedant` solves almost all of them.

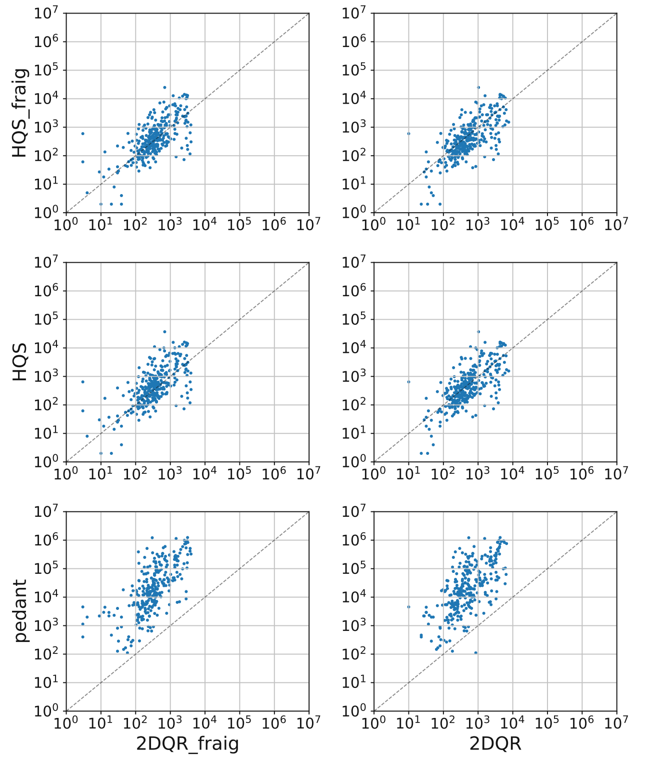
Additional remark: PEC (with 2 black boxes) and succinct 2-colorability are both PSPACE-complete [15, 29]. Thus, it is natural to ask if we can reduce them to QBF instances and use



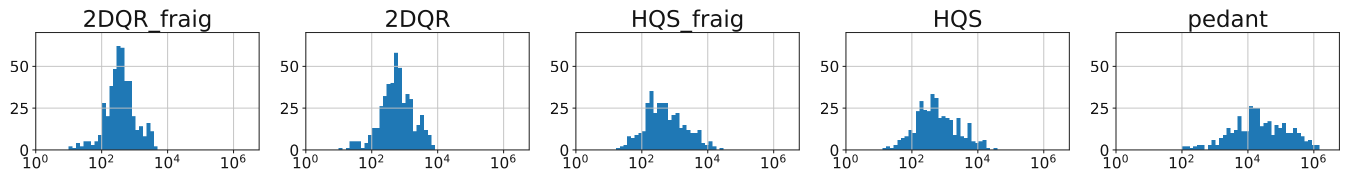
(a) Number of solves vs time in PEC instances. The black horizontal dotted line indicates the number of instances.



(b) Scatter plot for the time needed on each instance.



(c) Scatter plot for the number of AND gates in the Skolem function on each test case. `fraig` denotes the usage of `fraig` during post processing. The axes are the number of AND gates.



(d) Histogram for the number of AND gates in the Skolem function of each method. The x-axis is the number of AND gates and the y-axis is the number of instances.

Fig. 4: Various plots for PEC instances.

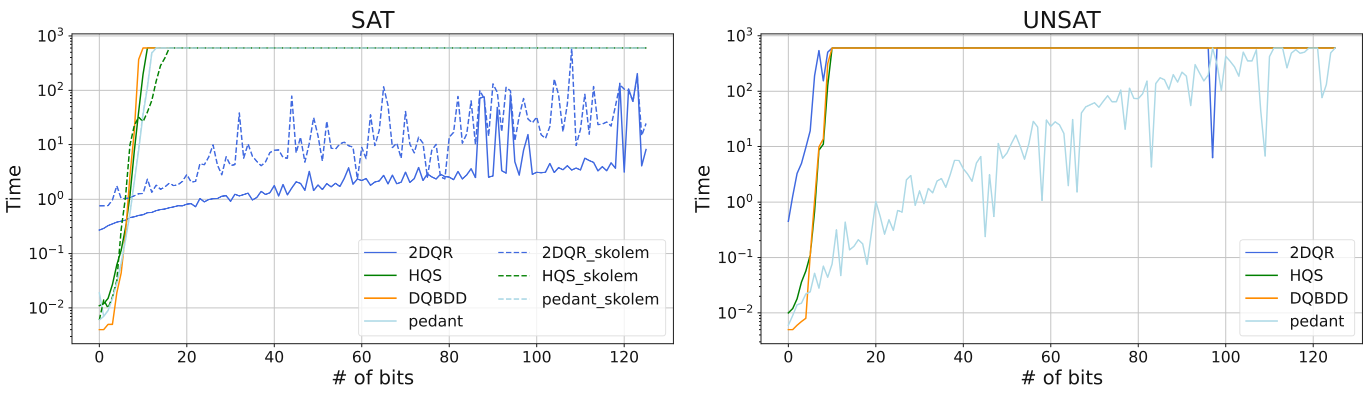


Fig. 5: Time needed to solve for n-bit 2-colorability test cases.

QBF solvers to solve them. In all instances QBF solver time out. This is not surprising since the only known reduction to QBF is the one in [3] which yield quadratic blow-up in the number of variables. For example, in the PEC instances with n universal variables, the resulting QBF would have about $24(n+2)^2$ variables. Our smallest PEC instance already uses 24 universal variables, and the resulting QBF would have at least 16000 variables, which is beyond the capability of current QBF solvers.

V. Conclusions

We introduce a novel technique to use PDR algorithms to solve 2-DQBF instances. The main insight is based on the properties that 2-DQBF is essentially a succinct 2-CNF formula. We also give a method for extracting both the positive and negative certificates for 2-DQBF based on the certificates provided by the PDR solver. We implement our reduction with `AVR` as the PDR solver and empirically show that this approach performs better than the state-of-the-art DQBF solvers on most of the PEC and 2-colorability test cases with a very large margin, except `Pedant` on some unsatisfiable 2-colorability test cases.

We believe our work is just the tip of the iceberg. First, we note that the Skolem function generation could be improved if we use an incremental approach, i.e., by modifying `AVR` to support incremental solving like [30]. Another direction is to efficiently integrate our 2-DQBF solver inside a general DQBF solver, which is very similar in spirit to how modern SAT solvers utilize 2-SAT solvers, e.g., when applying the Unit Propagation strategy, the SAT solver inadvertently is solving 2-SAT instances. We leave this as future work.

Acknowledgements

We acknowledge the generous funding from the National Science and Technology Council of Taiwan under grant NSTC 111-2923-E-002-013-MY3 and the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-113L900903.

References

[1] G. Peterson, J. Reif, and S. Azhar. “Lower bounds for multiplayer noncooperative games of incomplete information”. In: *Computers & Mathematics with Applications* 41.7 (2001), pp. 957–992.
[2] V. Balabanov, H.-J. K. Chiang, and J.-H. R. Jiang. “Henkin quantifiers and Boolean formulae: A certification perspective of DQBF”. In: *Theoretical Computer Science* 523 (2014), pp. 86–100.
[3] L. J. Stockmeyer. “The polynomial-time hierarchy”. In: *Theoretical Computer Science* 3.1 (1976), pp. 1–22.
[4] L. Henkin. “Some Remarks on Infinitely Long Formulas”. In: *Journal of Symbolic Logic* 30.1 (1961), pp. 167–183.
[5] C. Scholl and B. Becker. “Checking equivalence for partial implementations”. In: *Design Automation Conference (DAC)*. 2001, pp. 238–243.
[6] K. Gitina et al. “Equivalence checking of partial designs using dependency quantified Boolean formulae”. In: *International Conference on Computer Design (ICCD)*. 2013, pp. 396–403.
[7] R. Bloem, R. Könighofer, and M. Seidl. “SAT-Based Synthesis Methods for Safety Specs”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2014, pp. 1–20.
[8] J.-H. R. Jiang, V. N. Kravets, and N.-Z. Lee. “Engineering Change Order for Combinational and Sequential Design Rectification”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020, pp. 726–731.
[9] K. Chatterjee et al. “Distributed synthesis for LTL fragments”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2013, pp. 18–25.
[10] R. Wimmer et al. “From DQBF to QBF by Dependency Elimination”. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2017, pp. 326–343.
[11] F. Reichl, F. Slivovsky, and S. Szeider. “Certified DQBF Solving by Definition Extraction”. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2021, pp. 499–517.
[12] F. Reichl and F. Slivovsky. “Pedant: A Certifying DQBF Solver”. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2022, 20:1–20:10.
[13] J. Síc and J. Strejcek. “DQBDD: An Efficient BDD-Based DQBF Solver”. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2021, pp. 535–544.
[14] C. Scholl et al. “A PSPACE Subclass of Dependency Quantified Boolean Formulas and Its Effective Solving”. In: *AAAI Conference on Artificial Intelligence (AAAI)*. 2019, pp. 1584–1591.

[15] L. Fung and T. Tan. "On the Complexity of k-DQBF". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2023, 10:1–10:15.

[16] A. Biere, C. Artho, and V. Schuppan. "Liveness Checking as Safety Checking". In: *Electronic Notes in Theoretical Computer Science* 66.2 (2002), pp. 160–177.

[17] A. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2011, pp. 70–87.

[18] N. Eén, A. Mishchenko, and R. Brayton. "Efficient Implementation of Property Directed Reachability". In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 125–134.

[19] A. Goel and K. Sakallah. "AVR: Abstractly Verifying Reachability". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2020, pp. 413–422.

[20] U. Bubeck. "Model-Based Transformations for Quantified Boolean Formulas". PhD thesis. University of Paderborn, 2010.

[21] A. Fröhlich et al. "iDQ: Instantiation-Based DQBF Solving". In: *Pragmatics of SAT Workshop (POS)*. 2014.

[22] V. Balabanov and J. R. Jiang. "Reducing Satisfiability and Reachability to DQBF". In: *QBF Workshop*. 2015.

[23] A. Bradley and Z. Manna. "Checking Safety by Inductive Generalization of Counterexamples to Induction". In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2007, pp. 173–180.

[24] T. Seufert et al. "Everything You Always Wanted to Know About Generalization of Proof Obligations in PDR". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 42.4 (2023), pp. 1351–1364.

[25] F. Brglez, D. Bryan, and K. Kozminski. "Combinational profiles of sequential benchmark circuits". In: *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*. 1989, pp. 1929–1934.

[26] R. Brayton and A. Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *International Conference on Computer Aided Verification (CAV)*. 2010, pp. 24–40.

[27] H. Galperin and A. Wigderson. "Succinct Representations of Graphs". In: *Inf. Control.* 56.3 (1983), pp. 183–198.

[28] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008, pp. 337–340.

[29] C. Papadimitriou and M. Yannakakis. "A Note on Succinct Representations of Graphs". In: *Inf. Control.* 71.3 (1986), pp. 181–185.

[30] M. Blankestijn and A. Laarman. "Incremental Property Directed Reachability". In: *Formal Methods and Software Engineering*. 2023, pp. 208–227.

Projective Model Counting for IP Addresses in Access Control Policies

Loris D'Antoni, Andrew Gacek, Amit Goel, Dejan Jovanović,
Rami Gökhan Kıcı, Dan Peebles, Neha Rungta, Yasmine Sharoda, Chungha Sung
Amazon Web Services
Seattle, USA
{lorisd,gacek,amgoel,dejajov,ramikici,dgp,rungta,sharoday,chunghs}@amazon.com

***Abstract*—Zelkova is an AWS service that answers questions about Identity and Access Management (IAM) access policies such as "Does this policy allow *public* access?". Zelkova formalizes IAM policies and the meaning of "public" as a logical query that can be solved using SMT solvers. Among other conditions, Zelkova defines a policy as *public* if it allows access from a number of IP addresses that exceeds a given threshold. Encoding this check so that it is supported by all SMT solvers in the Zelkova portfolio is difficult because counting and restricting the number of models are not core SMT features. We describe two SMT encodings for checking whether the number of IPs allowed by a policy exceeds a given bound. Both encodings generate an SMT formula that can be discharged with a single call to an off-the-shelf SMT solver. Our approach takes less than 3s to detect whether a policy is public for 99.999% of the evaluated policies.**

I. Introduction

Millions of customers use AWS to store their data in a variety of resources such as databases and key-value stores. These resources are secure-by-default and accessible only when the customer grants access. The customer can grant access by authoring policies in the Identity and Access Management (IAM) language. The IAM language can express properties varying from simple sharing to complex constraints with a logical combination of positive and negative operators.

AWS offers tools to help customers write and understand their policies. One of these tools is Block Public Access (BPA) [9] which protects customers from accidentally attaching "public" policies to their resources.

The central design decision in BPA is the exact definition of "public", and three factors are at play here. First, the definition must match a customer's intuitions about public access. Second, the definition must be mathematically precise so it can be checked in a provable way. Specifically, a precise mathematical definition of public access allows us to check whether a policy is public using Zelkova [6], an IAM policy analysis service based on SMT solvers. Third, the definition must require no additional information from the customer so that BPA itself can be a one-click solution.

The key idea underlying the AWS definition of public is to examine a customer policy and extract out the trusted entities—e.g. individual account IDs, networks, or users. In general, a policy should only reference a limited number of trusted entities. If there is any access granted outside this small set of trusted entities, e.g., due to misuse of wildcards—e.g., a policy that allows access from any account ID—the policy is considered to be granting public access. However, when reasoning about IP addresses, treating each IP as a separate trusted entity may make a policy look public when it really is not. A single customer may own a large collection of IP addresses, all of which are grouped together and considered trusted—e.g., the company may own the `19.0.0/8` range of IP addresses. This Classless Inter-Domain Routing (CIDR) notation represents the set of $2^{24} \approx 1.7 \times 10^6$ IP addresses between `19.0.0.0` and `19.255.255.255` inclusively. A single customer policy could reference hundreds of similarly sized CIDR blocks, which in total amounts to granting access to the entire internet. Such a policy should be considered public, so we need to update our definition of BPA to handle IP addresses. For the domain of IP addresses we have decided to draw a line at a specific number of IP addresses that can be allowed before a policy is considered public. From conversations with customers, we identified that any number of IP addresses larger than a single `/8` CIDR block—i.e., 2^{24} IP addresses—should be considered public.

To check for public access, Zelkova turns an IAM policy into a logical query that is discharged using a portfolio of SMT solvers. The original version of the BPA check [9] removes all trusted parts of the policy and then compiles the remaining parts into a single logical formula. If an SMT solver finds a model for the generated formula, the policy allows untrusted access and is marked public. To support IP addresses as trusted entities, we must precisely count how many IP addresses are allowed by the policy after the other trusted parts are removed.

Today, there are no SMT solvers that support precise model counting and that can solve this problem within a few seconds [19]. Therefore, we encode this **bounded projective IP-counting problem** into a single SMT query using arithmetic. The key idea of our encoding is to split the set of IP addresses into equivalence classes for which counting is trivial and then reduce the bounded projective IP-counting problem to an SMT query that checks if the sum of the IP counts in the "allowed" equivalence classes exceeds the given bound. We also present an encoding that eliminates the need for arithmetic by pre-computing minimal sets of summands which will exceed the bound. These two encodings allow us to solve the bounded projective IP-counting problem using existing SMT solvers

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_26

with the time constraints imposed by AWS customer needs (i.e., within 3 seconds per check). We remark that both encodings use a *single* SMT query and do not rely on features such as incremental solving of multiple SMT queries (e.g., one per equivalence class), which may result in expensive enumeration and are not necessarily supported by all SMT solvers.

Contributions: This paper makes three contributions:

We formalize the bounded projective IP-counting problem (Section II) and illustrate how Block Public Access needs to solve this problem (Section III).

We design two sound and efficient SMT encodings for solving the bounded projective IP-counting problem (Section IV). The encodings compute equivalence classes of IP addresses and separately compute the sizes of each equivalence class, thus bypassing the need to reason about individual IP addresses and the need for model counting. The first encoding, which is supported by all but one of the SMT solvers in the Zelkova portfolio, requires support for arithmetic operations (e.g., summing) on top of the theories already required by Zelkova. The second uses a knapsack-based approach to identify combinations of equivalence classes that can exceed the threshold and avoids arithmetic operations, thus imposing no additional requirements on the SMT solver. This last encoding is supported by all SMT solvers in the Zelkova portfolio.

We evaluate the encodings on 700,000 policies containing IP addresses; our approaches take less than 3s (the time limit required by the target application) to detect whether a policy is public for 99.999% of the evaluated policies (Section V).

II. The Bounded Projective IP-counting Problem

In this section, we first describe the AWS policy language and its semantics, and then define the problem of checking whether the number of IP addresses for which at least one request is allowed by a given policy exceeds a given bound.

The AWS policy language is defined as serialized JSON [1]. In this paper, we describe a simplified abstract syntax of the core constructs of the language to simplify our exposition. As done in prior work [9], we model an IAM policy as a set of statements that can either allow or deny a set of requests. A request is granted when it is allowed by at least one statement and not denied by any statement. In the rest of the section, we formalize these concepts.

Requests. We assume a set of variables V, which represent the possible fields in a request—e.g., `principal`, `action`, `resource`, and `sourceIP` are variables.

A request $r : V \to$ `val` is a function that maps a variable to its value (the value can be `null`). For example, the partial snippet of a request r_1 shown in Figure 1 maps the variable `principal` to the value `111122223333:user/Bob`, the variable `action` to the value `s3:ListBucket`, the variable `resource` to the value `bucket/invoices`, and the variable `sourceIP` to the value `20.121.201.3`. Each variable $v \in V$ is associated with a value of a specific type in IAM and we use $\tau(v)$ to denote it. Common value types are booleans, strings, and IP addresses. Less common types are integers

```
r1:(principal: 111122223333:user/Bob,
    action   : s3:ListBucket,
    resource : bucket/invoices,
    sourceIP : 20.121.201.3,
    username : Bob, ...)
```

Fig. 1: A request allowed by statement s_3 in Figure 2.

and floats. Every request contains the variables `principal`, `action`, and `resource`, whereas others are optional. Requests might not contain values for all the variables, so we allow r to map variables to the special value `null`.

Statements. A statement s is a pair (e, Ψ) where e is either the value `allow` (we call these statements *allow-statements*) or the value `deny` (we call these statements *deny-statements*), and $\Psi : V \mapsto$ `pred` is a partial function that maps variables to predicates. For example, in the statement $s_3 = ($`allow`$, \Psi_3)$ in Figure 2, Ψ_3 maps the variable `action` to the predicate stating that the action of a request should start with `s3:` (i.e., the predicate is represented by the pattern `s3:*`). Common predicate types are simplified regular expressions to restrict values of strings, boolean comparisons, and Classless Inter-Domain Routing (CIDR) [3] descriptions of sets of IP addresses. For example, the IP range `20.0.0.0/7` allows the $2^{32-7} = 33,554,432$ IPv4 addresses in the range `20.0.0.0` to `21.255.255.255`[1], which also includes the IP address `20.121.201.3` from the request r_1 in Figure 1.

We use $V(s)$ to denote the set of variables in the domain of Ψ—i.e., all the values for which the partial function is defined. Every statement always maps the variables `principal`, `action`, and `resource` to a predicate.

Intuitively, a statement matches a request if, for every variable appearing in the statement, the request's values are models of the corresponding predicates in the statement.

Definition (Statement-Matching Requests): *Given a request r and a statement $s = (e, \Psi)$ we say that s matches the request r if and only if, for every $v \in V(s)$, the value $r(v)$ is a model of the predicate $\Psi[v]$. We write $M(s)$ to denote the set of all requests matched by s—i.e, $M(s) = \{r \mid \bigwedge_{v \in V(s)} \Psi[v](r(v))\}$.*

The statement $s_3 = ($`allow`$, \Psi_3)$ in Figure 2 has five keys `principal`, `action`, `resource`, `sourceIP`, and `username` and matches the request r_1 in Figure 1.

We simplify the syntax of IAM policies and assume that each key is associated with its predicate. Our implementation maps the JSON representation of statements to this predicate format; this translation is straightforward and syntax-directed and we do not present it formally here.

Policies. A policy $P = \{s_1, s_2, ..., s_n\}$ is a set of statements and we use $AS(P)$ (resp. $DS(P)$) to denote all the allow (resp. deny) statements in P. We write $P = (AS(P), DS(P))$

[1]We note that some of the IP addresses in this range are not usable, e.g., `20.0.0.0` and `21.255.255.255`, but in this paper we assume the size of a CIDR also considers unusable IP addresses.

```
s1:(allow,  (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 201.0.0.0/7,
             account   : 444455556666))
- - - - - - - - - - - - - - - - - - - - - - -
s2:(allow,  (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 14.0.0.0/7,
             username  : Alice))
- - - - - - - - - - - - - - - - - - - - - - -
s3:(allow,  (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 20.0.0.0/7,
             username  : Bob))
- - - - - - - - - - - - - - - - - - - - - - -
s4:(deny,   (principal: *,
             action    : s3:*,
             resource  : bucket/*,
             sourceIP  : 14.0.0.0/8))
```

Fig. 2: An IAM policy with three allow statements s_1, s_2, s_3 and one deny statement s_4. The predicates in green describe IP ranges, and the predicates `14.0.0.0/7` and `14.0.0.0/8` describe overlapping sets of IP addresses.

to directly denote these two sets and a and d instead of s to denote an allow or deny statement respectively.

Definition (Granted Requests): *A policy P grants a request r if and only if there exists an allow statement that matches the request, i.e., $\exists a \in AS(P).\ r \in M(a)$, and there does not exist a deny statement that matches the request, i.e., $\forall d \in DS(P).\ r \notin M(d)$. We write $Granted(P)$ to denote the set of all requests granted by the policy P, which can be defined as follows.*

$$Granted(P) = (\bigcup_{a \in AS(P)} M(a)) \setminus (\bigcup_{d \in DS(P)} M(d))$$

The policy depicted in Figure 2 has three allow statements and one deny statement and allows the request r_1 shown in Figure 1, which only matches statement s_3.

Problem Definition. Given a policy P we define the set of IPs for which at least one request is granted by P as the set $IPSet(P) = \{r(\texttt{sourceIP}) \mid r \in Granted(P)\}$. We are now ready to define the problem solved in this paper.

Definition (Bounded Projective IP-Counting Problem): *The bounded projective IP-counting problem is to determine if the number of IP addresses from which at least one request is allowed by a policy P exceeds a given IP-count threshold τ, which formally can be stated as $|IPSet(P)| > \tau$.*

III. Illustrative Example

We illustrate our approach for solving the IP-count bounding problem using the example policy presented in Figure 2. This policy is for an Simple Storage Service (S3) bucket [2], an object storage service, and it consists of four statements s_1, s_2, s_3 and s_4.

The first statement s_1 grants access for anyone from account `444455556666` to perform S3 actions on any S3 object in the bucket `bucket`. The second statement s_2 grants access for a user named `Alice` to perform S3 actions on any S3 object in the bucket `bucket`. Similarly, the third statement s_3 grants access for a user named `Bob` to perform S3 actions on the same objects. Because users work in different companies, the statements allow requests from different ranges of IP addresses for each user (denoted in green).

The fourth statement s_4 is a deny-statement that removes access for any requests coming from an IP in the range `14.0.0.0/8`. This IP range is a subset of the IP range `14.0.0.0/7` allowed by statement s_2.

The question we are interested in answering is whether the policy in Figure 2 allows *public* access to the S3 bucket. As discussed in Section I, some AWS customers want to consider a resource to be publicly accessible if the number of IP addresses from which one can issue an allowed request exceeds a given threshold. However, not all IPs should contribute to the total count. In our example, the statement s_1 only allows requests from a specific account ID. Because account IDs are assigned by AWS (unlike usernames), this statement is already associated with what in Section I we called a trusted entity and it is irrelevant how many IP addresses it allows access from. The existing work on AWS public access [9] can detect such trusted entities and remove this statement from the policy before we need to reason about IP addresses.

Once statement s_1 has been removed, we are ready to count how many IP addresses the other statements allow requests from. In this section, we assume that the threshold is $\tau = 2^{24} = 16,777,216$ IP addresses—i.e., the size of a single `/8` CIDR block, which is the largest block size owned by a single entity. In general, the threshold can be set to any value.

Checking whether the number of IP addresses exceeds the threshold requires *counting* how many IP addresses one can issue an allowed request from, a problem that on the surface requires going beyond the capability of SMT solvers, the current tool of choice for reasoning about public access in IAM policies [9], [6]. The encodings proposed in this paper provide a way of checking if the number of allowed IP addresses exceeds the threshold τ using traditional SMT solvers (i.e., without counting models).

We discuss what are the key insights of the encoding.

IP Equivalence Classes: The SMT encoding used by Zelkova to describe what requests each statement allows (or denies) is a conjunction of monadic predicates[2] where each predicate describes what values a request can contain for each specific variable, and particularly for source IPs. For example, the statement s_2 is translated by Zelkova into the following SMT formula φ_{s_2} involving the theory of Strings (e.g., $L(R)$ denotes the language of a regular expression R),

[2]We use the term monadic for predicates that involve one variable—e.g., the predicate $x > 0$ is monadic, whereas the predicate $x > y$ is polyadic.

and bit-vectors (e.g., $\mathit{in\_ip\_range}(\texttt{sourceIP}, \texttt{I})$ denotes a predicate for checking if the value of variable sourceIP is in the set of bit-vectors encoding IPs belonging to the CIDR block I):

$$\begin{gathered}\texttt{principal} \in L(\star) \wedge \texttt{action} \in L(\texttt{s3:}\star)\wedge \\ \texttt{resource} \in L(\texttt{bucket/}\star)\wedge \\ \mathit{in\_ip\_range}(\texttt{sourceIP}, \texttt{20.0.0.0/7})\wedge \\ \texttt{username} = \text{“}\texttt{Alice}\text{”}\end{gathered} \tag{1}$$

The final result describing whether a request is allowed by the policy is expressed by the formula $\varphi_P(\bar{x}, \texttt{sourceIP}) = (\varphi_{s_1} \vee \varphi_{s_2} \vee \varphi_{s_3}) \wedge \neg\varphi_{s_4}$, that is, a request is allowed by the policy if it is allowed by an allow statement and not denied by any deny statement. The notation $\varphi_P(\bar{x}, \texttt{sourceIP})$ separates the variable sourceIP from all other free variables—i.e., $\bar{x}$. Our goal is to check whether $\#SAT(\exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{sourceIP})) > 2^{24}$. Because the source IP predicates are all monadic (i.e., they do not interact with other variables other than sourceIP), the predicate $\varphi_P(\bar{x}, \texttt{sourceIP})$ can be expressed as a Boolean combination of predicates of the following form (where i denotes the ith statement):

$$\psi_i^1(\bar{x}) \wedge \psi_i^2(\texttt{sourceIP})$$

Therefore, our first key idea is that we can take all satisfiable Boolean combinations of the predicates $\psi_i^2(\texttt{sourceIP})$—i.e., all the predicates of the form $\mathit{in\_ip\_range}(\texttt{sourceIP}, \texttt{range})$—to obtain IP predicates describing equivalence classes of IP addresses—i.e., if a request is allowed (resp. denied) with an IP address, replacing that address with another one in the same class will still make the request allowed (resp. denied).

We show in Section IV-B how to compute equivalence, but in our example, after removing s_1, we have 3 equivalence classes (other combinations are unsatisfiable and we simplify each predicate $\mathit{in\_ip\_range}(\texttt{sourceIP}, \texttt{range})$ as simply its range range):

- $e_1 = \texttt{14.0.0.0/8}$ is the result of the Boolean combination $\texttt{14.0.0.0/7} \wedge \neg \texttt{20.0.0.0/7} \wedge \texttt{14.0.0.0/8}$;
- $e_2 = \texttt{20.0.0.0/7}$ is the result of the Boolean combination $\neg \texttt{14.0.0.0/7} \wedge \texttt{20.0.0.0/7} \wedge \neg \texttt{14.0.0.0/8}$;
- $e_3 = \texttt{15.0.0.0/8}$ is the result of the Boolean combination $\texttt{14.0.0.0/7} \wedge \neg \texttt{20.0.0.0/7} \wedge \neg \texttt{14.0.0.0/8}$.

In our example, each predicate is defined using a single CIDR block, but in general, a predicate can be described as a union of CIDR blocks; we support this more general form in our approach presented in Section IV.

Counting IPs without Model Counting: Once we have computed equivalence classes, it is trivial to compute how many IP addresses each class contains using the definition of a CIDR block.

- $|e_1| = |\texttt{14.0.0.0/8}| = 2^{24} = 16,777,216$;
- $|e_2| = |\texttt{20.0.0.0/7}| = 2^{25} = 33,554,432$; and
- $|e_3| = |\texttt{15.0.0.0/8}| = 2^{24} = 16,777,216$.

With this information available, we can now write an SMT formula that checks whether the sum of allowed IPs exceeds the threshold. (We write $\varphi_P(\bar{x}, \texttt{ip})$ to denote the result of substituting the variable sourceIP with the constant ip in the formula φ_P).

$$\begin{aligned}&(\text{if } \exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{14.0.0.0}) \text{ then } 2^{24} \text{ else } 0) \ + \\ &(\text{if } \exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{20.0.0.0}) \text{ then } 2^{25} \text{ else } 0) \ + \\ &(\text{if } \exists\bar{x}.\ \varphi_P(\bar{x}, \texttt{15.0.0.0}) \text{ then } 2^{24} \text{ else } 0) > 2^{24}\end{aligned} \tag{2}$$

Intuitively, each row uses the formula φ_P to check if the policy allows a representative IP address from each equivalence class, in which case it contributes the size of that class to the counter. In this case, the policy is public because it allows access from $2^{24} + 2^{25} > 2^{24}$ IP addresses.

The constraint in (2) requires arithmetic to describe whether the number of IP addresses exceeds the threshold. Because some SMT solvers do not support both the theories of strings and arithmetic at the same time [21], in Section IV we also introduce a version of the encoding that pre-computes what combinations of equivalence classes can exceed that given threshold and generates a formula that does not involve arithmetic. In our example, the minimal combinations of equivalence classes that exceed the threshold are $|e_1| + |e_3|$ and $|e_2|$, therefore one can write an SMT formula that does not use arithmetic and that checks whether the sum of allowed IPs exceeds the threshold as follows:

$$\begin{gathered}(\varphi_P(\bar{x}, \texttt{14.0.0.0}) \wedge \varphi_P(\bar{x}, \texttt{15.0.0.0})) \ \vee \\ \varphi_P(\bar{x}, \texttt{20.0.0.0})\end{gathered} \tag{3}$$

In this case, the formula is satisfied by making the second disjunct true, thus denoting that the 2^{25} IP addresses allowed by the equivalence class e_2 exceed the threshold 2^{24}.

IV. Counting IPs without Model Counting

Before presenting our technique for solving the bounded projective IP-counting problem presented in Section II, we recall that our goal is to devise an SMT-based approach for solving the problem that does not rely on model counting.

At the high-level, given a policy P and a threshold τ our main approach proceeds in two steps:

1) We compute a set of equivalence classes of IP addresses such that all the IP addresses appearing in the same equivalence class e are treated the same way by the policy P (Section IV-B)—i.e., if a request r with an IP address in e is granted (resp. not granted) by the policy P, the request obtained by replacing the IP address with any member of the equivalence class e is still granted (resp. not granted).
2) Once the equivalence classes are computed, we can separately compute the size of each equivalence class (i.e., the number of IP addresses in it), and rewrite the SMT formula encoding the policy semantics to remove any mention of IP addresses and instead directly reason about the size of each equivalence class (Section IV-C).

Step 2 in the algorithm above requires arithmetic operations, and for some solvers, specifically NFA2SAT [21], this theory is not supported in combination with the many theories (e.g., strings) required to model IAM policies. To address this limitation, we introduce a new encoding that avoids arithmetic and instead uses a knapsack-based approach to compute a new formula that identifies combinations of equivalence classes that can lead to exceeding the threshold. The formula is entirely expressible in propositional logic (Section IV-D).

Before presenting our approaches, we distill the essence of the problem solved by our approach to a purely logical formalization that is agnostic from the specific problem of counting IPs (Section IV-A).

A. Bounded Projective Counting Problem

Prior work on verifying policies in the IAM language [9], [6] has shown how, given a policy P, one can create a formula $\varphi_P(x_1, \ldots, x_n)$ that is satisfied *exactly* by all the granted requests in the set $Granted(P)$. Specifically, each variable x_i corresponds to a variable in the set V, and a satisfying assignment $c_1, \ldots, c_n$ corresponds to the request mapping each variable to the corresponding value—i.e., $[x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$.

Thanks to the above formalization if we consider `sourceIP` to denote the variable denoting a source IP address in a request, by appropriately massaging the formula φ, we can express the bounded projective IP-counting problem defined in Section II as a formula of the following form

$$\#SAT(\exists \bar{x}.\ \varphi_P(\bar{x}, \texttt{sourceIP})) > \tau \tag{4}$$

Here $\#SAT(\cdot)$ is the function denoting the number of satisfying assignments to a formula. Because `sourceIP` is the only non-quantified (i.e., free) variable, the set of satisfying assignments to the formula $\exists \bar{x}.\ \varphi_P(\bar{x}, \texttt{sourceIP})$ corresponds exactly to the set $IPSet(P)$. Thus, Equation (4) correctly captures the bounded projective IP-counting problem.

With this observation, we can focus the rest of the section on the following generalized version of the counting problem.

Definition (Bounded Projective Counting Problem): *We say that a formula $\varphi(x, y)$ exceeds a y-count threshold τ if the following is true:*

$$\#SAT(\exists x.\ \varphi(x, y)) > \tau \tag{5}$$

In the rest of the section, we show how one can avoid solving the hard $\#SAT(\cdot)$ problem over the quantified formula $\exists x.\ \varphi(x, y)$ by instead solving an easier satisfiability problem over formulas involving only y.

B. Computing Equivalence Classes

Given a formula of the form $\exists x.\ \varphi(x, y)$, the first step of our algorithm is to compute equivalence classes for the variable y for the following equivalence relation, which captures that two values of y are equivalent if they behave the same for every possible value of x. Because computing maximal equivalence classes is in general unnecessary and in fact something we want to avoid (as we will see later), we instead define valid partitions of the domain into equivalent elements.

Definition (y-equivalence, y-partition): *Given a formula $\varphi(x, y)$ say that two constants c_1 and c_2 are y-equivalent iff*

$$\forall x.\ \varphi(x, c_1) \iff \varphi(x, c_2).$$

*We say a partition $\Pi = \{e_1, \ldots, e_j\}$ forms a y-*partition *of the domain $Dom(y)$ with respect to y-equivalence iff (i) Π is a valid partition of $Dom(y)$ (i.e., the union of all e_i is $Dom(y)$, and all elements of Π are disjoint), and (ii) for every class $e_i \in \Pi$, all elements of e_i are y-equivalent.*

If the variable y only appears within monadic predicates in the formula $\varphi(x, y)$ (which is the case for the problem of IP-count bounding), we can always compute a y-partition of $Dom(y)$ by computing the set of minimal satisfiable Boolean combinations of all the monadic predicates over y, also called minterms [16].

For example, if the only predicates involving y in the formula $\varphi(x, y)$ are the monadic predicates $\psi_1(y)$ and $\psi_2(y)$, a valid y-partition can be computed as the set

$$\Pi = \{\psi_1 \wedge \psi_2, \neg\psi_1 \wedge \psi_2, \psi_1 \wedge \neg\psi_2, \neg\psi_1 \wedge \neg\psi_2\}$$

If any of the predicates in Π is unsatisfiable, they can be discarded before continuing to the next steps. In the worst case, the y-partition can contain exponentially many classes in the size of the formula $\varphi(x, y)$, but in practice this is rarely the case.

The appealing aspect of computing y-partitions in the aforementioned way is that one *does not* need to reason about satisfiability of the whole formula $\varphi(x, y)$ and instead only needs to check satisfiability of Boolean combinations of predicates involving y, which in our application domain, counting IPs, is a very friendly theory to work with, as we illustrate next.

`sourceIP`*-equivalence:* For IP addresses, each monadic predicate appearing in an IAM statement is a union of CIDR blocks $c_1 \cup \ldots \cup c_n$ (in our running example, each union only contain one CIDR block). We note that two CIDR blocks can be disjoint, or one can be a subset of the other; other logical relations are not possible—e.g., partial overlap. We can therefore assume that $c_1 \cup \ldots \cup c_n$ contains all disjoint CIDR blocks (ones that are subsets of others can be removed).

After this pre-processing, by collecting positive and negative terms, any satisfiable Boolean combination of unions of CIDR blocks can be written in the following form:

$$(\psi_1 \cap \ldots \cap \psi_j) \setminus (\psi_{j+1} \cup \ldots \cup \psi_k).$$

The right-hand side of the $\setminus$ is itself a union of CIDR blocks.

We next show that the left-hand side can also be rewritten as a union of CIDR blocks and that the $\setminus$ of two unions of CIDR blocks can also be translated to a union of CIDR blocks.

Given two unions of CIDR blocks $C_1 \cup \ldots \cup C_n$ and $D_1 \cup \ldots \cup D_m$, their *intersection* can be defined as the union of CIDR blocks $\cup_{i \le n, j \le m} C_i \cap D_j$, where the intersection of two CIDR blocks is defined as:

1) $C \cap D = \emptyset$ if C is disjoint from D;
2) `IP1/M1` $\cap$ `IP2/M2` $=$ `IP2/M2` $\cap$ `IP1/M1` $=$ `IP1/M1` if `IP1/M1` is a subset of `IP2/M2`—i.e., `M1` $\geq$ `M2` and the first `M2` bits of `IP1` and `IP2` are the same.

Given two unions of CIDR blocks $C_1 \cup \ldots \cup C_n$ and $D_1 \cup \ldots \cup D_m$, their *difference* can be defined as $\cup_{i \le n} (\cap_{j \le m} C_i \setminus D_j)$ where

1) $C \setminus D = \emptyset$ if $C \subseteq D$;
2) $C \setminus D = C$ if $C \cap D = \emptyset$;
3) if `IP1/M1` $\supset$ `IP2/M2`—i.e., `M1` $<$ `M2` and the first `M1` bits of `IP1` and `IP2` are the same—we recursively split the CIDR of `IP1/M1` into two longer CIDR blocks (the one obtained by choosing the (`M1`+1)-th bit to be 0 or 1, respectively) and recursively subtract `IP2/M2` from them—i.e., `IP1/M1` $\cap$ `IP2/M2` $=$ (`IP1[1..M1]0/M1+1` $\setminus$ `IP2/M2`) $\cup$ (`IP1[1..M1]1/M1+1` $\setminus$ `IP2/M2`).

Because of the 0-1 splitting in case 3, the above algorithm guarantees that CIDR blocks appearing in the final union are all disjoint.

Example (From Section III): *In the example in Section III, the class* e_3 $=$ `15.0.0.0/8` *is the result of* `14.0.0.0/7` $\setminus$ (`20.0.0.0/7` $\cup$ `14.0.0.0/8`). *First, we rewrite the formula as* (`14.0.0.0/7` $\setminus$ `20.0.0.0/7`) $\cap$ (`14.0.0.0/7`$\setminus$`14.0.0.0/8`)) *following the definition of* $\setminus$ *on unions of CIDR blocks. The first conjunct is* `14.0.0.0/7` *following case 2 of the definition of* $\setminus$ *on CIDR blocks, whereas the second conjunct is rewritten as* (`14.0.0.0/8`$\setminus$ `14.0.0.0/8`)$\cup$(`15.0.0.0/8`$\setminus$`14.0.0.0/8`)) *following case 3. Now, the first disjunct rewrites to the empty set (case 1), and the second disjunct rewrites to* `15.0.0.0/8` *(case 2). Finally,* $e_3 =$ `14.0.0.0/7`$\cap$`15.0.0.0/8` $=$ `15.0.0.0/8`.

C. Arithmetic Approach

The arithmetic approach formalizes the IP counting problem as a summation problem. Recall that our goal is to asses whether the formula in Equation (5) is true. One way to encode this problem as an SMT formula involving arithmetic operations for counting is to rewrite the formula as follows:

$$(\Sigma_{c \in Dom(y)} \text{ if } \exists x.\ \varphi(x, c) \text{ then } 1 \text{ else } 0) > \tau \qquad (6)$$

By skolemizing the existentially quantified variable x, we can simplify the formula as follows:

$$(\Sigma_{c \in Dom(y)} \text{ if } \varphi(x_c, c) \text{ then } 1 \text{ else } 0) > \tau \qquad (7)$$

The encoding in Equation (6) is expressible as an SMT formula whenever $Dom(y)$, the domain of y, is finite. However, if the domain of y is large (which is the case when y represents IP addresses) solving Equation (6) will either require a very large SMT formula or iterating over many possible smaller formulas (one per IP address).

We call this approach the Arithmetic Approach (AA). Our Arithmetic Approach sidesteps this problem thanks to the previously computed equivalence classes, which we call $EC_y(\varphi(x, y))$). For every equivalence class e in the set $EC_y(\varphi(x, y))$, we use the symbol rep_e to denote a representative value of y from that class. Equation (7) can then be optimized as the following formula:

$$(\Sigma_{e \in EC_y(\varphi(x,y))} \text{ if } \varphi(x_e, rep_e) \text{ then } |e| \text{ else } 0) > \tau \qquad (8)$$

If we consider the example formula in Equation (1), and the equivalence class e_1 $=$ `14.0.0.0/8` the formula $\varphi(x_{e_1},$ `14.0.0.0`$)$ can be obtained by replacing the variable x with x_{e_1} and the variable `sourceIP` with the concrete IP `14.0.0.0`. The encoding in Equation (8) is expressible as an SMT formula whenever the size $|e|$ of an equivalence class e is computable.

Because our y-partition algorithm computes equivalence class that are expressed as monadic predicates $\psi(y)$, all one needs to generate the formula in Equation (8) is a technique for counting the number of models for the theory of y, a trivial problem for predicates involving IPs.

Theorem (Soundness of Arithmetic Approach): *A formula* $\varphi(x, y)$ *exceeds a* y*-count threshold* τ *iff Equation* (8) *holds.*

Proof. We know that for any two elements c_1, c_2 in the same y-equivalence class $e \in EC_y(\varphi(x, y))$, the following holds $\forall x.\ \varphi(x, c_1) \iff \varphi(x, c_2)$. Thus, $\exists x.\ \varphi(x, rep_e)$ holds iff $\exists x.\ \varphi(x, c)$ holds for every $c \in e$. Therefore, the formula if $\exists x.\ \varphi(x, rep_e)$ then $|e|$ else 0 correctly computes the size of the equivalence class e. □

Note that if the formula $\varphi(x, y)$ lies in a theory $\mathcal{T}$, the constraints in Equation (8) are in the theory $\mathcal{T}$ + QFLIA.

Counting IPs: In particular, when y represents IP addresses, $|e|$ can be computed efficiently. If e is represented by a set of disjoint CIDR blocks—which the Boolean operations defined in Section IV-B guarantee—then $|e|$ is the sum of the size of each CIDR block. In particular, for IPv4, the size of a CIDR `IP/M` is $2^{32-\texttt{M}}$—e.g., |`14.0.0.0/8`| $= 2^{24}$.

D. Arithmetic-free Approach

The arithmetic approach discussed in Section IV-C requires adding an arithmetic theory on top of the theory $\mathcal{T}$ needed to reason about the formula $\varphi(x, y)$. In some cases, an SMT solver might support the theory $\mathcal{T}$, but not the combined theory, e.g., $\mathcal{T}$ + QFLIA. For example, the NFA2SAT solver [21] is a powerful solver used by Zelkova [6] to prove properties of policies, and relies on SAT solving to reason about strings and does not support arithmetic. Since industrial applications rely on portfolio solving to provide performance and robustness, an ideal solution to the bounded projective IP-counting problem should work with all possible available solvers.

In this section, we describe an approach for solving the y-count bounding problem entirely within the theory $\mathcal{T}$—i.e., without the need for an arithmetic theory for counting. We call this approach the Arithmetic-free Approach (AFA).

At a high level, the AFA proceeds in the following steps:

1) First, it uses a dynamic programming algorithm (a variant of knapsack) to compute *all minimal combinations of equivalence classes* $\mathcal{C}$ that can cause the threshold τ to be exceeded.
2) Then, it creates a new constraint $\psi_{\mathcal{C}}$ that is satisfied exactly by combinations of equivalence classes that exceed the threshold τ.

Computing Minimal Possible Violations: We have shown in Section IV-C how we can compute the size $|e|$ of every equivalence class e. First, we define the combinations of equivalence classes that are minimal possible violations of the given threshold. Given a set A of equivalence classes, we write $Weight(A)$ to denote the sum of the sizes of the equivalence classes in A—i.e., $Weight(A) = \sum_{e \in A} |e|$.

Definition (Minimal Possible Violation): *Given a set of equivalence classes EC, we say that a subset $A \subseteq EC$ forms a* possible violation *of the threshold τ iff the sum of the sizes of each class exceeds the threshold—i.e., $Weight(A) > \tau$.*

Furthermore, A is a minimal possible violation *iff no strict subset of A is a possible violation—i.e., $\neg\exists e \in A.\ Weight(A) \geq Weight(A \setminus \{e\}) > \tau$.*

In the worst case, if we have n equivalence classes, it is possible to have $2^{O(n)}$ minimal possible violations. For example, if we have a threshold of $n/2$ and each class has weight 1, there are approximately $\binom{n}{n/2}$ minimal possible violations—i.e., all ways to pick $n/2$ classes from the set. In practice, the number of minimal possible violations is often much smaller as illustrated by the following example.

Example (Minimal IP Violations): *We discussed in Section III how the example in Figure 2 leads to three equivalence classes,* `14.0.0.0/8`, `20.0.0.0/7`, *and* `15.0.0.0/8`.

The minimal possible violations are obtained by the sets {`14.0.0.0/8`, `15.0.0.0/8`} *and* {`20.0.0.0/7`}.

The problem of computing the set of all minimal possible violations can be solved using a variant of the knapsack dynamic programming algorithm. Intuitively, starting from an empty set, one can build incrementally larger subsets, by adding additional equivalence classes as long as the unused equivalence classes can still be used to cross the threshold. By considering the equivalence classes ordered by their size, this process ensures that we can stop as soon as we cross the threshold, resulting in a minimal possible violation.

Minimal Satisfiable Violations: We now assume we have computed the set $MPV = \{A_1, \ldots, A_m\}$ of all minimal possible violations. The last step is to find one that is an actual satisfiable violation—i.e., a set A_i such that each class $e \in A$ makes the formula $\exists x.\ \varphi(x, rep_e)$ true.

To encode this problem as a constraint, we introduce for each class e, a new variable v_e to model whether the class e corresponds to a positive class (i.e., one that makes the formula $\exists x.\ \varphi(x, rep_e)$ true) or a negative class (i.e., one that makes the formula $\exists x.\ \varphi(x, rep_e)$ false). After replacing the existentially quantified variable x with x_e, we get the constraint:

$$v_e \Leftrightarrow \varphi(x_e, rep_e) \tag{9}$$

The y-count bounding problem can then be solved by checking satisfiability of the following formula, which simply looks for a minimal possible violation $A \in MPV$ consisting only of positive classes.

$$\bigvee_{A \in MPV} \bigwedge_{e \in A} v_e \tag{10}$$

For our example in Section III, we obtain Equation (3).

	Arithmetic		Arithmetic-free	
	# fastest	% fastest	# fastest	% fastest
cvc4	2,012	0.3%	403	0.1%
cvc5	196,431	28.0%	92,412	13.2%
trivial	500,927	71.6%	500,996	71.5%
z3	622	0.1%	241	0.1%
nfa2sat	N/A	N/A	105,948	15.1%
timeout (3s)	13	0.0%	7	0.0%

TABLE I: The numbers of problems solved by every solver in the Zelkova portfolio.

Theorem (Soundness of Arithmetic-free Approach): *A formula $\varphi(x, y)$ exceeds a y-count threshold τ iff the conjunction of the constraints in Equations* (9) *and* (10) *is satisfiable.*

Proof. In Equation (9), variable y_e can only be true if every $\exists x.\ \varphi(x, rep_e)$ holds. From the definition of y-equivalence, we have that $\exists x.\ \varphi(x, rep_e)$ holds iff $\exists x.\ \varphi(x, c)$ holds for every $c \in e$. Therefore Equation (10) is true iff and only if there exists a combination of minimal possible violations that is actually satisfiable. The definition of MPV and minimal satisfiable violation ensures that if any violation exists—i.e., there exists a set of equivalence classes that exceeds the threshold—there also exists a minimal satisfiable version of it that is considered in Equation (10). □

If $\varphi(x, y)$ lies in a theory $\mathcal{T}$, assuming a decision procedure for counting models over each equivalence class e, the constraints in Equations (9) and (10) are in the theory $\mathcal{T}$.

V. Implementation and Evaluation

The Block Public Access (BPA) feature detects if a bucket is publicly accessible (`Public`) or not (`Not Public`), and is integrated in many AWS services. Some services use BPA as a preventative control that prevents attaching any policy that is detected to be public to an AWS resource. BPA is an essential guard rail to ensure data is not exposed to broad access. Other detective services, like Config, Macie, Guard Duty and Security Hub, reports to customers which resources have public policies attached, without preventing any access.

Implementation: BPA is built on top of Zelkova's encoding of IAM policies and runs on the portfolio of solvers supported by Zelkova. Zelkova runs on AWS Lambda, a serverless computing platform that runs applications without users needing to provision or manage servers. Zelkova currently uses the solvers CVC4, CVC5, Z3, and NFA2SAT [21] as part of its portfolio. The arithmetic-free approach is supported by all solvers, whereas the arithmetic approach produces an encoding that is not supported by NFA2SAT [21]. Zelkova invokes all supported solvers in parallel and returns the results as soon as one of the solvers provides the answer.

Evaluation: We evaluate the performance of our encodings on 700K randomly chosen policies that contain IP addresses and set a timeout of 3 seconds. The BPA checker [9] performs a pre-processing step that simplifies statements that do not allow any access to untrusted entities. This step removes a large number of statements, thus leaving us with 225,215 policies to still analyze with our technique. We report a timeout if none of the solvers terminates within 3s, the timeout used in production for BPA checks. We run our experiments on an x86_64 cloud desktop running Amazon Linux version 2 with 96 CPUs and 382GB memory.

Table I shows how many times (# fastest) and for what percentage of the benchmarks (% fastest) each solver was the fastest. The arithmetic approach described (Section IV-C) times out on 13 policies whereas the arithmetic-free approach (Section IV-D) times out on 7 policies. The arithmetic-free approach could solve 6/13 problems the arithmetic approach timed out on, whereas the arithmetic approach couldn't solve any of the policies the arithmetic-free approach timed out on.

The average running time of the arithmetic approach is 8ms, with 50% of the policies terminating within 3ms, 90% of the policies terminating within 21ms, and 99.99% of the policies terminating within 850ms. The average running time of the arithmetic-free approach is 8ms, with 50% of the policies terminating within 3ms, 90% of the policies terminating within 21ms, and 99.99% of the policies terminating within 709ms.

The arithmetic-free approach is on average 0.09 times slower (geomean) than arithmetic approach. However, it times out on 6 fewer policies.

The arithmetic-free approach enables using the NFA2SAT solver [21], and for 105,948 queries (15.13% of our dataset) the NFA2SAT was faster than any other solver. Table I presents the number of problems solved by each solver in the Zelkova portfolio for both techniques.

To summarize, both solving approaches are effective for the BPA application and the arithmetic-free approach can solve more queries, but is slightly (0.09 times) slower.

We further analyze the experiments. The formula size ranges from 2K to 1,095K bytes (avg. 17K), the number of equivalence classes ranges from 1 to 9 (avg. 1.1), the time taken to compute equivalence classes ranges from 1ms to 254ms (avg. 2ms), and the SMT solver takes 1ms to 1,559ms (avg. 8ms). The time taken to compute the MPV sets in the arithmetic-free approach ranges from 1ms to 50ms (avg. 1ms). The additional data indicates that the SMT solvers takes most of the time needed to check BPA for IPs.

VI. Related Work

Previous work on Block Public Access [9] did not address the issue of counting IP addresses; this new use case emerged afterwards through conversations with customers. The previous work was designed for cases where there were a relatively small number of trusted values drawn from an overwhelmingly larger universe of possibilities. The difference in size between the trusted values and the possible universe meant that no model counting was necessary; any set of trusted values was small enough. For IP addresses, one needs to consider large sets of trusted values drawn from the limited universe of IP addresses. Here one must count models to precisely capture the boundary between public and non-public access.

Quacky [17] can quantify the permissions provided by IAM policies. It uses model counting to count how many requests of size up to a certain bound a policy can match. While Quacky solves a different problem, their methodology could be in principle adapted to count how many IPs one can issues requests from. As we have argued, model counting is a feature supported by very few solvers; Quacky uses one solver called ABC [5], which only supports strings and integer constraints (and thus limited sets of policies). Furthermore, model counting is generally expensive: for simple EC2 policies consisting of often just one statement, Quacky incurs an average running time of more than 100s, a time that is not acceptable for customers using BPA. We attempted using the solver `abc` used by Quacky [17] to quantify permissions of access control policies, but `abc` did not support the constraints generated by Zelkova, specifically the theory of bit vectors.

To put other work in context, we will consider them through the lens of the requirements of Zelkova. Zelkova is a live AWS service handling customer access policies and supporting many security use cases where soundness is paramount.

Model Counting in SAT: Model counting and model enumeration [19] are established research areas in the SAT community, with a wide range of application in domains that require quantitative analysis—e.g., probabilistic inference [13]. Approaches from SAT that directly enumerate and count Boolean-level models have straightforward translations to the SMT level [20], and are available in some SMT solvers [15]. In our domain, it is insufficient to enumerate the Boolean structure without accounting for the complex SMT formulas involving, e.g., strings, generated by Zelkova to model IAM policies. Enumerating all models is also infeasible due to the size of the solution space—i.e., a typical threshold for the IP-count bounding problem is 2^{24}!

Approximate Model Counting: To avoid enumeration of all models in SAT, approximate model counting [12] relies uses universal hashing to provide provable approximations. Beyond the initial theoretical results [23], this approach yielded highly scalable tools for SAT problems [11], [18] with extensions to extended to some SMT theories, such as bit-vectors [10] and linear arithmetic [14]. These results do not work in the presence of string constraints, which are ubiquitous when modeling IAM policies. Most importantly, our application domain—i.e., counting IPs—requires exact results and cannot rely on probabilistic approximations.

Model Counting in SMT: Precise model counting at the SMT level has mostly focused on individual theories such as integers [8] and restricted theory of strings [22]. Our domain requires reasoning about many string operations, often combined with the theories of arithmetic over integers. The most relevant work that attempts to cover these theories translates string and integer constraints into automata representations that facilitates counting of feasible solutions [4], [5]. Because this line of work relies on transforming constraints to automata, it is limited to string constraints with integer bounds and thus cannot handle the full multi-sorted constraints required by Zelkova.

Summary: In contrast to related work, our approach encodes the counting constraints into a single SMT query that relies on standard SMT-LIB [7] language and theories. The use of standard SMT language allows one to rely on battle-tested general-purpose SMT solvers that match Zelkova's portfolio approach and goals.

VII. Conclusion

This paper defined the IP-count bounding problem as the problem of checking whether the number of IPs from which an IAM policy allows requests exceeds a given bound. The bounded projective IP-count problem is formalized logically as the *bounded projective counting problem* where the goal is to check whether a formula $\#SAT(\exists x.\ \varphi(x, y)) > \tau$ is true.

We presented two SMT encodings of the bounded projective counting problem that avoid the need to solve a model counting problem—i.e., the $\#SAT(\cdot)$ primitive—for which no performant solver support exists. Our encodings are general: if the variable y only appears within monadic predicates in the formula $\varphi(x, y)$ and one has access to a model counter for the theory of y (but one for the theory of x is not required!), our encodings generate SMT formulas that are true iff the bounded projective counting problem admits a solution.

The generality of our encoding opens opportunities to solve other bounding problems for IAM policies, but also in other domains, e.g., if constraints denote valid tuples in a table and one wants to bound the number satisfying assignments for the values of a numerical column.

References

[1] Amazon Web Services. Amazon IAM, Apr 2024. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

[2] Amazon Web Services. Amazon S3, Apr 2024. URL: https://aws.amazon.com/s3/.

[3] Amazon Web Services. What is CIDR?, Apr 2024. URL: https://aws.amazon.com/what-is/cidr/.

[4] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272. Springer, 2015.

[5] A. Aydin, W. Eiers, L. Bang, T. Brennan, M. Gavrilov, T. Bultan, and F. Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 400–410, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236024.3236064.

[6] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018.

[7] C. Barrett, A. Stump, C. Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[8] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, 1994.

[9] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, et al. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 281–291, 2020.

[10] S. Chakraborty, K. Meel, R. Mistry, and M. Vardi. Approximate probabilistic inference via word-level counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[11] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings 19*, pages 200–216. Springer, 2013.

[12] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Approximate model counting. In *Handbook of Satisfiability*, pages 1015–1045. IOS Press, 2021.

[13] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008. URL: https://www.sciencedirect.com/science/article/pii/S0004370207001889, doi:10.1016/j.artint.2007.11.002.

[14] D. Chistikov, R. Dimitrova, and R. Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pages 320–334. Springer, 2015.

[15] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[16] L. D'Antoni and M. Veanes. Automata modulo theories. *Commun. ACM*, 64(5):86–95, 2021. doi:10.1145/3419404.

[17] W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince, and T. Bultan. Quacky: Quantitative access control permissiveness analyzer. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022.

[18] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *International Conference on Machine Learning*, pages 334–342. PMLR, 2013.

[19] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of satisfiability*, pages 993–1014. IOS press, 2021.

[20] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, pages 424–437. Springer, 2006.

[21] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka. Solving string constraints using SAT. In C. Enea and A. Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 187–208. Springer, 2023. doi:10.1007/978-3-031-37703-7_9.

[22] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 565–576, 2014.

[23] L. Stockmeyer. The complexity of approximate counting. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 118–126, 1983.

Toward Exhaustive Sequential Redundancy Removal

Rohit Dureja*, Jason Baumgartner*, Raj Kumar Gajavelly*, Robert Kanzelman*, and Kristin Y. Rozier†
*IBM Corporation †Iowa State University

***Abstract*—Hardware designs often contain logical redundancies: pairs of behaviorally-equivalent gates. *Sequential redundancy removal* is the process of removing gates that are behaviorally-equivalent within the reachable states of a design. It has many applications in the hardware design process, including logic optimization, equivalence checking, accelerating functional verification, and engineering change-order optimization.**

Redundancy removal is an intricate process, orchestrating various algorithms to compute equivalence-classes of potentially-equivalent gates, then to prove their validity. In this paper, we introduce techniques to enable *exhaustive* redundancy removal on practical designs, such as resource-balancing the underlying algorithms; self-tailoring them to sequentially-deep logic; and detailing an orders-of-magnitude optimization to the *Proof Graph* essential to proving non-inductive redundancies. We integrate these techniques within a state-of-the-art redundancy removal framework, illustrating their efficacy on various benchmarks.

I. Introduction

Hardware designs are often rife with logical redundancies. Some are deliberate, e.g. to improve circuit timing or implement error-resilience features. Many are unexpected and undesired; including them in semiconductor devices degrades cost and circuit performance, and increases power consumption.

In verification, logical redundancies are even more prevalent, e.g. due to input constraints disabling various functionality, and redundancies arising between design and testbench logic. *Equivalence checking* (EC) and *engineering change order* (ECO) tools compare two related designs; significant redundancy is common between those designs. Redundancy removal is highly-beneficial to verification scalability, solving some properties outright [1]–[3]; is the core solving procedure of EC [4, 5]; and can yield smaller ECOs [6].

Sequential redundancy removal frameworks (Fig. 1) identify, then eliminate, functionally-equivalent gates. Each suspected redundancy requires proving a property, called a *miter*, confirming that a pair of gates behave identically in the reachable states of a design. Simulation is used to refine incorrect equivalence-classes of gates, correcting inaccurate miters [4, 7]. Once a miter is proven, design size and power can be reduced by replacing one of its gates by the other [5]. The choice of which gate to eliminate can be delay- and placement-aware yielding higher-performance circuits.

Many techniques have been proposed to accelerate redundancy removal. For example: combinational redundancy removal solves miters from topologically-shallowest to deepest, reducing effort for deeper miters by leveraging early-merging and prior refinements [8, 9]. *Speculative reduction* models assumptions through structural logic simplifications, enabling a *transformation-based verification* (TBV) suite of model-checking algorithms to benefit from those assumptions to solve the non-inductive miters [2]. A *Proof Graph* enables early-merging of selective miters even before a fixedpoint of all-miters-proven is achieved, minimizing the number of proofs necessary to converge, and yielding reductions even if a resource-limit precludes convergence [10].

Contributions: We introduce various improvements to sequential redundancy removal in the pursuit of *exhaustiveness*. **(1)** We present *sequential resource-sweeping* (Sec. III-A) to self-tailor SAT-based bounded model-checking (BMC) [11] and induction to the sequential depth of the design, enabling them to solve deeper miters. This yields $\approx 5\%$ greater redundancy removal in less runtime via induction, and $\approx 20\%$ fewer incorrect miters deferred to TBV. **(2)** We propose techniques to balance counterexample simulation runtime with solving effort (Sec. III-B), yielding $\approx$ 30% overall speedup (Sec. III-B). **(3)** We address scalability challenges of deep-counterexample generation and simulation, via: separate *eager shallow* vs. *lazy deep simulation* phases to accelerate $\approx 16\%$ additional deep–logic refinement (Sec. III-C); obtaining $\approx 34\%$ complementary deep refinements via seeded-state BMC (Sec. III-D); and minimally-lossy techniques to approximate pathologically-deep miters impractical to simulate (Sec. III-E). **(4)** We present a near-linear-runtime algorithm to construct a *Proof Graph* [10], improving scalability by orders of magnitude (Sec. III-F). Experiments in Sec. IV show our techniques yielding $2.1\times$ speedup to EC, 32.4% speedup with 16.9% more solves in model-checking, and enabling *exhaustive* redundancy removal on netlists up to 857110 AND gates, 75952 registers.

II. Preliminaries and Related Work

We represent a hardware design as a *netlist* N, comprising a directed graph $G = \langle V, E\rangle$. Vertices V represent logic gates of different types: constants, primary inputs, combinational primitives such as AND gates, and sequential primitives such as *registers*. Edges $E \subseteq V \times V$ represent interconnections between gates. The *fanin* (*fanout*) of gate u is the set of gates reachable by traversing edges backward (forward) from u. A *strongly-connected component* (SCC) is a set of gates having a directed path between every pair of gates within the SCC.

Registers have *initial values* defining their time-0 behavior, and *next-state functions* defining their time-$i{+}1$ behavior. A *trace* is a sequence of Boolean valuations to gates over timesteps, beginning from an *initial state* consistent with initial-values at time 0. A *state* is a Boolean valuation to the registers; a *reachable state* is one reachable along a trace. Certain gates may be labeled as *properties*, representing a verification objective to obtain a *counterexample trace* illustrating an assertion of that gate, or to prove the absence of any

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_27

```
exhaustive_sequential_redundancy_removal (Netlist $N$)
1: Create equivalence-classes of gates in $N$, where gate $u$ in class $Q(u)$
   is suspected functionally-equivalent to every other gate in $Q(u)$.
2: Select a representative gate $R(Q(u))$ from each class $Q(u)$.
3: Construct speculatively-reduced netlist $N'$ (§II-A2) replacing the
   source gate $u$ of each $(u, v) \in E$ by $R(Q(u))$; else copy $N' = N$.
4: For each gate $v$, add a miter to $N'$ falsified when $v \not\equiv R(q(v))$.
5: Construct a Proof Graph $P$ from $N'$ and the set of miters $M$.
6: Attempt to falsify or prove each miter (§II-A1).
7: If $P$ was computed or speculative-reduction was not used, merge the
   soundly-proven-miter gates onto their representatives (§II-A3).
8: If a miter was not proven, refine equivalence classes to separate their
   gates (simulating available counterexamples, §II-A4); goto Step 2.
9: Merge proven miter gates onto their representatives.
```

Fig. 1. Exhaustive sequential redundancy removal algorithm.

counterexample. During redundancy removal, properties called *miters* are created confirming the equivalence of postulated gate redundancies. Each miter is represented as an XOR over the suspected gate equivalence. Certain gates may be labeled as *observables*, e.g. primary outputs of the design. A *merge* of gate u onto gate v consists of moving output edges of u onto v, then deleting u. Any gate not in the fanin of a property or observable is irrelevant to netlist behavior, outside of its *cone of influence* (COI). After merging a pair of redundant gates, a secondary set of gates may become irrelevant in this way.

A. Redundancy Removal

Fig. 1 shows an exhaustive-capable sequential redundancy removal algorithm. It first overapproximates the redundancy candidates, represented as *equivalence classes* of gates suspected as pairwise functionally-equivalent in all reachable states. Initial equivalence-classes are constructed via: **(1)** run-time options and syntactic information, e.g. whether to compute redundancies only over registers vs. all gate types, and whether to pre-filter e.g. via *corresponded signal name pairs* in equivalence-checking; **(2)** compatible random-simulation signatures [4, 7]. (While not depicted for brevity, redundancy-removal can efficiently be performed *modulo inversion*.)

The *miter* properties M are then created and solved, comparing each gate to its equivalence-class representative. If a miter is falsified, its counterexample shows a miscompare of the corresponding gates (and/or fanin gates, due to *speculative reduction*). Simulating a counterexample on the original netlist often refines additional incorrect equivalence-classes. If a miter is unsolved due to resource limits or incomplete proof technique, it must be pessimistically pruned from its equivalence-class. If *all* miters are proven, the equivalence-classes represent valid redundancies; the netlist may be optimized by *merging* equivalent gate-pairs. Else the process repeats until this fixed-point is achieved, or global timeout.

1) Induction: k-Induction [12] is commonly used to prove miters [1, 4]. The *base case* is bounded model checking (BMC), validating miters during the first *k* timesteps from the initial states. The *inductive step* verifies miters in *k* timesteps from any state (reachable or not) which satisfies the set of

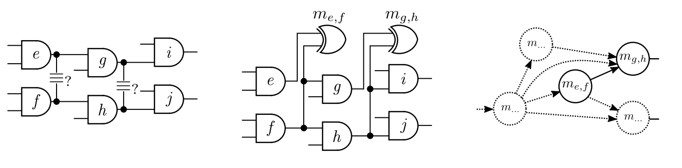


Fig. 2. Assumption modeling for scalable redundancy removal. (a) Suspected redundancies over gates. (b) Speculatively-reduced netlist with miters over suspected redundancies. (c) Proof Graph representing miter dependencies.

mutually-postulated equivalences within fewer timesteps. Using a sufficiently-large k and unique-state constraints, induction can be a *complete* proof technique [1]. Though BMC and induction solve NP-complete problems and become unscalable as k increases, rendering them *incomplete* in practice.

Redundancy removal of large netlists often requires solving millions of miters, partially because each gate may be compared to a sequence of varying representatives across refinements. BMC can efficiently falsify, and *k*-induction can efficiently prove, many miters. Though practical netlists often comprise non-inductive miters. Discarding even a single non-inductive yet accurate miter precludes *exhaustive* redundancy removal, and often causes an iterative avalanche of fanout miters to become unscalable as equivalence-classes are refined. Stronger algorithms (logic reduction, abstraction, alternate proof and falsification techniques) are thus useful to solve the non-inductive miters to enable exhaustiveness [2, 3].

2) Speculative reduction: An *assume-then-prove* paradigm enables assuming certain miters when proving others [1, 4]. *Speculative reduction* [2] models assumptions by replacing the fanout edge from each gate to an edge from its equivalence-class representative, aside from input edges to the miters themselves (necessary for soundness). Fig. 2a shows suspected-redundant gate pairs $\{e, f\}$ and $\{g, h\}$; Fig. 2b shows the speculatively-reduced netlist with miters $m_{e,f}$ and $m_{g,h}$.

Speculative reduction reduces the amount of logic in the fanin of each miter, making many inductive and often yielding orders of magnitude speedup. However, speculatively-reducing an incorrect miter alters fanout behavior of its non-representative *speculatively-reduced gate*, i.e. e and g in Fig. 2b. Therefore, this technique hinders the ability to early-merge a proven miter's gates until *all of its fanin speculatively-reduced gates* have been proven accurate. Therefore, speculative reduction may require a fixedpoint of proving all remaining miters, before any merge can be safely performed.

3) Early merging: A *Proof Graph* records the set of miters whose speculatively-reduced gates may alter the behavior of fanout miters, allowing to *early-merge* certain proven miters before converging a fixedpoint [10]. The Proof Graph is a directed graph $P = \langle M, D\rangle$ with one vertex in M per miter, and edges $D \subseteq M \times M$ representing dependencies of fanout miters upon the speculative-reduced gate of each miter. Miter m_1 is dependent on miter m_2 if the speculatively-merged gate of m_2 is in the fanin of either gate of m_1. Fig. 2c shows a Proof Graph for the speculatively-reduced netlist of Fig. 2b.

A miter proof is *sound* when the miters of all of its fanin speculatively-merged gates are proven. When a miter is soundly proven, its gates may be safely merged, regardless of other falsified or unproven miters. Sound proofs are propagated through fanout nodes of the Proof Graph, which may enable proven fanout miters to become soundly-proven. For k-induction, the Proof Graph only needs to record miter-dependencies relevant to the k-timestep unrolled netlist, along with dynamically-added SAT proof dependencies upon postulated induction-hypothesis constraints. For TBV, *all* transitive fanin dependencies are recorded, to ensure soundness using arbitrary model-checking algorithms [10].

The Proof Graph does not alter the set of miters to be proven. It improves scalability by: **(1)** prioritizing miter-solution order, generalizing combinational topological ordering [8, 9] to cyclic sequential netlists. *Leaves* can be solved first, minimizing effort wasted solving possibly-unsound miters. With parallel orchestration, leaf miters can be stubbornly solved, in parallel to time-balanced iteration among others [13]. **(2)** More redundancy is identifiable before global timeout. **(3)** The number of times each miter is repeatedly solved across refinements is reduced. This is especially important when using stronger model-checking algorithms to solve non-inductive miters: a PSPACE-complete problem. **(4)** Early-merging within the original netlist yields other speedups, e.g. faster simulation, unrolling, SAT, and future Proof Graph reconstruction.

4) Trace simulation: When a miter is falsified, simulating its counterexample on the original netlist identifies a set of inaccurate miters. Failed induction proofs yield counterexamples starting from possibly-unreachable *induction leak* states; those may be simulated to refine other non-inductive miters.

Simulation can consume significant runtime, and thus requires careful orchestration. *Bit-parallel simulation* atomically simulates 64 independent patterns in each 64-bit machine word. Counterexamples from BMC, induction, and TBV may be accumulated into machine words [7, 14], allowing each simulation to refine multiple counterexamples. Additional proposed improvements include packing *compatible* counterexamples into the same machine-word pattern [14]; randomizing unimportant input values; and permuting copies of counterexamples across patterns, e.g. with distance-1 modifications [15].

With shallow analysis, e.g. k-induction with small k, each miter affects only a local fanout region. This allows to decompose the netlist into slightly-overlapping components to be analyzed in parallel using fixed-depth simulation [5, 16, 17]. *Exhaustive* redundancy removal *additionally* requires deep analysis across more timesteps. As the depth of analysis increases, the inter-dependence of miters extends toward the entire cone-of-influence; windowing becomes ineffective, and simulation of the sequential netlist becomes inevitable.

III. Exhaustive Redundancy Removal

We describe the main contributions and experimentally evaluate their isolated impact on exhaustive redundancy removal

```
sequential_resource_sweeping (Netlist $N$, Miters $M$)
Miters $M_u := \emptyset$, $M_i := \emptyset$, $M_s := \emptyset$  # sets of miters
for $k \in 0, 1, 2, 3, \ldots$ :  # iterate over increasing k-depth
    for satLimit $\in$ min, ..., max :  # iterate over SAT limits
        # run BMC to increase bounded-proof depth $\geq k$
        $\langle$proved, falsified, unsolved$\rangle$ := BaseCase($k$, $M$, $N$, satLimit)  # BMC
          ↪ Check all miters with proof-depth $< k$ using BMC
          ↪ Update proof-depth of newly-BMC-proved
          ↪ Simulate falsified miters to refine equivalence-classes
        $M := M \setminus$ falsified  # discard BMC-falsified miters
        $M_u := M_u \setminus \{$ proved $\cup$ falsified $\}$  # update unsolved miters
        if satLimit $\equiv$ max : $M_u := M_u \cup$ unsolved  # cache unsolved miters
        # run induction on miters adequately checked by BMC
        $M_i := M$ # snapshot active-miters for later rollback from induction leaks
        if $M_s \not\equiv \emptyset$ :
            $M := M_s \cup$ newly-BMC-proved, $M_s := \emptyset$  # restore snapshotted miters
        else
            $M := M \setminus$ (miters with proof-depth $< k$)  # base-case inconclusive
            $M := M \setminus M_u$  # drop miters unsolved in prior induction steps
        repeat # fixedpoint (FP) iterations
            $N'$ := SpecReduce($N$, $M$), Graph $G$ := ProofGraph($N'$, $M$)
            $\langle$proved, falsified, unsolved$\rangle$ := InductiveStep($k$, $G$, $N'$, satLimit)
              ↪ Check miters in leaves of Proof Graph $G$
              ↪ Update Proof Graph for proven miters  # enable early-merging
              ↪ Simulate falsified miters to refine equivalence-classes
            $N$ := EarlyMerge($N$, $G$) and remove merged miters from $M$, $M_i$
            $M := M \setminus$ falsified  # drop falsified miters (induction leaks)
            if satLimit $\equiv$ max : $M_u := M_u \cup$ unsolved  # cache unsolved miters
            else if $|$unsolved$| > 0$ and $M_s \equiv \emptyset$ :  # FP iteration with unsolved
                $M_s := M$  # snapshot miters for next SAT iteration
            $M := M \setminus$ unsolved  # drop inconclusive miters
        until fixedpoint (no unsolved or falsified) for $k$ at satLimit
        $M := M_i$  # restore active-miter snapshot to roll-back induction leaks
    if $n$-steps with no merging or timeout : break  # self-tailor depth
```

Fig. 3. Sequential resource-sweeping using BMC and k-induction

in this section. End-to-end experimental results for various formal applications appear in Section IV.

A. Sequential resource-sweeping

Most miters are easy to solve at shallow BMC or induction depth, becoming unscalable as depth increases. Because satisfiability checking is NP-complete, some miters are pathologically-difficult, even at shallow depth. Large netlists often contain a diversity of logic, often comprising a mix of easier and difficult miters.

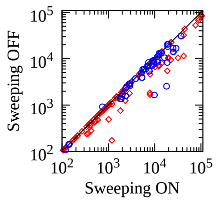
Borrowing from combinational equivalence checking [8, 9], sequential redundancy removal frameworks typically solve unfolded miters from topologically-shallower to deeper. Shallow miters are often easier; their solution simplifies fanout miters through merging unfolded gates, and refining incorrect equivalence-classes. Resource-limits may be applied, and another fixedpoint iteration attempted after refining unsolved miters. Due to diversity of miter-difficulty, combinational netlist simplification via BDD- and SAT-sweeping may benefit from iterating unsolved miters with increasing resource-limits, solving gradually-more-difficult miters without indefinite delays caused by pathological miters [7, 9].

For *exhaustive* sequential redundancy removal, additional resource-sweeping controls and equivalence-class management are necessary across k values, and to optimally defer miters into TBV. We introduce *sequential resource-sweeping* in Fig. 3 to manage these intricacies. For each k-depth, each miter is checked using increasing SAT-resource limits (lines 3–32). The induction base-case is validated by BMC (lines 5–8). The miters falsified with BMC are inaccurate, and permanently discarded (line 9). Any miter unsolved by BMC is skipped for that resource-limit, as induction would be unsound (line 17). The remaining miters are checked using induction (lines 19–31). Miters are solved from topologically-shallowest to deepest, using an inductive Proof Graph [10]; early-merging of soundly-proven miters is performed after iterating its leaves (line 25), offering runtime benefits (Sec. II-A3). Any miter unproven by induction is discarded (line 26 and 30). Because k-induction is incomplete, it may *spuriously* falsify accurate miters. For exhaustiveness, it is thus essential to snapshot and restore *active* miters M_i (line 13 and 32, resp.) to prevent induction leaks from permanently discarding accurate miters.

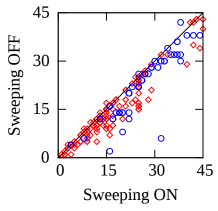
After completing a lossy fixedpoint, SAT resource is incremented up to a configurable maximum value, and previously-unsolved miters M_s are snapshotted to check anew (line 29). Any miter unsolved by the maximum SAT-limit during BMC (line 11) or induction (line 27) is deferred to TBV (line 18). Miters are then restored from M_s if available, incrementally reusing prior effort (line 15). Note that any proofs obtained using a subset of miters are valid for a superset. Thus induction-*proved* results when M_s was snapshotted may be re-applied after restoring, despite adding any newly-BMC-*proved* miters. However, the induction-hypotheses upon which reused proofs relied must also be annotated onto the inductive Proof Graph. Prior refinements reflected in M_s may be pessimistic within the current SAT-limit and k if any newly-BMC-*proved* miters are added. While spuriously-refined miters will be checked at higher k values or by TBV, this temporary lossiness may be compensated for, by: **(i)** using a dedicated pre-induction BMC phase, which often helps overall scalability anyway; **(ii)** skipping the restoration of M_s if newly-BMC-*proved* is non-empty; or **(iii)** selectively restoring previously-falsified miters that share logic with newly-BMC-*proved* to check anew.

The algorithm terminates after n timesteps without conclusive result, or global timeout (line 33). *Exhaustive* redundancy removal often benefits from deeper BMC than induction, to minimize wasted effort trying to prove inaccurate miters and to accelerate early-merging; $n \geq 5$ for BMC, and $n \geq 2$ for induction, are used in all experiments. While many miters are sequentially shallow, robust redundancy removal requires self-tailoring to deeper netlists. Solving easier miters at greater BMC and induction depth is often faster than deferring those to TBV: inductive Proof Graphs have fewer dependencies, enabling earlier merging; counterexample generation for BMC requires less reconstruction effort than through a sequence of TBV engines. Thus it is beneficial to defer only the difficult miters to TBV, not the easier deep miters.

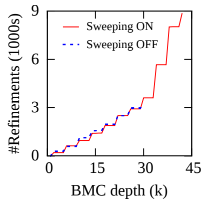
Most prior work imposes SAT resource-limits via bounding backtracks [9] or decisions [7]. While effective, those do not closely align with actual runtime. Time-limits are volatile and hurt reproducibility. We have found the number of propagations as a reproducible runtime-aligned metric, which also allows balancing simulation and solver runtime (Sec. III-B).



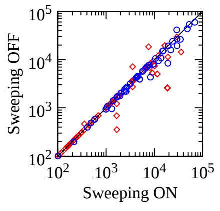
(a) #BMC refinements



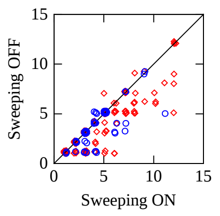
(b) BMC depth



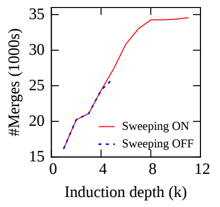
(c) #Refinements vs k



(d) #Induction merges



(e) Induction depth



(f) #Merges vs k

Fig. 4. Sequential resource-sweeping effectiveness: BMC and Induction for industrial logic optimization (○) and model-checking benchmarks [18] (◇)

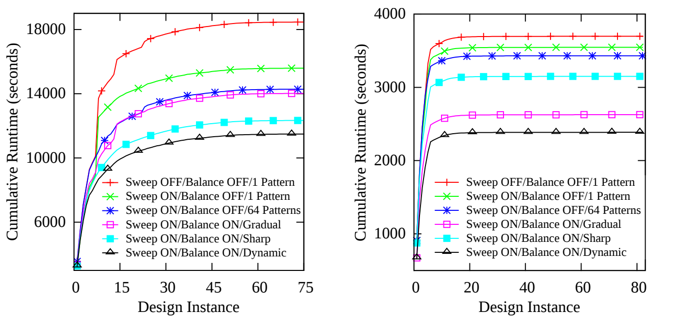
We evaluate sequential resource-sweeping in Fig. 4 for difficult industrial logic optimization and model-checking benchmarks [18]. Redundancy removal begins with a *refinement phase* of increasing-depth BMC, with 1800-second timeout early-terminated by 5 timesteps without refinement; then a *proof phase* of increasing-depth induction with 36000-second timeout early-terminated by 2 timesteps without merging.[1] SAT propagation-limits increment by $10\times$ from 10000 to 10000000 when resource-sweeping, and are unbounded otherwise. For industrial benchmarks, sequential resource-sweeping enables 30.3% deeper BMC on average (32.4% for model-checking) (Fig. 4b), yielding up to 20.7% more refinements (23.1% for model-checking) (Fig. 4a). It also enables deeper induction (Fig. 4e), yielding 5.2% more merged gates on average (3.4% for model-checking) (Fig. 4d). Note that unbounded SAT-resource can sometimes solve a *lucky miter* with modestly more than the maximum resource-sweeping SAT-limit, and thus occasional modest losses for resource-sweeping occur. These are offset by more-frequent, larger wins by deferring difficult-for-SAT miters to TBV, while enabling BMC and induction to solve easier deeper miters.

Fig. 4c and Fig. 4f show detailed per-k refinement and merging respectively for a single deep benchmark. BMC completes 12 more steps, and induction goes 6 steps deeper, with resource-sweeping enabled. While some common success is achieved at shallower k, self-tailoring depth when resource-sweeping yields 201.5% more refinements and 32.4% more merged gates at greater depth.

B. Resource-balanced simulation

Bit-parallel simulation allows atomically refining multiple counterexamples, packed into each bit of a machine word.

[1]The time-limits used in these experiments are rarely encountered, though are imposed for uncommon yet inevitable pathological scenarios.



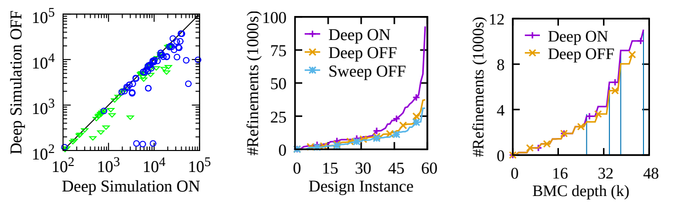
(a) SEC Benchmarks (b) Model-checking benchmarks [18]

Fig. 5. Cumulative runtime with resource-balanced simulation

Resource-sweeping ensures that easier miters are solved early, while difficult miters are deferred. When SAT-limits are small, miters are solved quickly: accumulating more traces before simulating is often faster. As SAT-limits increase, miter solving is slower: simulating fewer traces more-frequently is often faster. For greatest scalability, it thus is beneficial to tailor simulation frequency as miters are solved within the resource-sweeping loop of Fig. 3 (lines 8 and 24).

Predictive tuning of how many traces to accumulate vs. SAT-limit can be improved via *dynamic resource-balancing*, comparing runtime of the prior simulation to runtime of solvers since the last simulation. When either exceeds the other by more than a configurable threshold, simulation can be deferred or expedited. (For better reproducibility, runtime can be estimated by comparing number of gates and timesteps simulated vs. SAT-propagations since the last simulation.) While resource-sweeping ensures somewhat-balanced resource per miter, dynamic balancing adjusts for factors such as the percentage of miters proven vs. falsified vs. resource-exceeded, or unexpected solver-resource variance using less-predictable model-checking algorithms via TBV.

We evaluate resource-balancing in Fig. 5a on industrial sequential equivalence-checking (SEC) benchmarks, and Fig. 5b on model-checking benchmarks. SAT-limits increment by $10\times$ from 10000 to 1000000 propagations. Cumulative runtime is plotted for: **(1)** resource-sweeping disabled (baseline); simulation-resource balancing disabled with **(2)** each trace simulated vs. **(3)** 64 traces accumulated; resource-balancing enabled with **(4)** gradual tapering of 64 patterns at 10000 propagations, 32 at 100000, and 1 at 1000000, **(5)** sharp tapering from 64 patterns at 10000 propagations to 1 above; **(6)** dynamic runtime balancing. Note that unsolved benchmarks are not plotted, for clarity. Settings **(3)-(6)** are 8.38%, 10.07%, 20.87% and 27.28% faster, respectively, than **(2)** in Fig. 5a; vs. 3.24%, 25.92%, 11.18% and 32.71% in Fig. 5b. The reason that sharp tapering wins for SEC, and gradual for model-checking, is due to the percentage of miters falsified: corresponded signal-name pairing for SEC ensures fewer incorrect non-inductive miters. Dynamic balancing adjusts effectively to either scenario. Setting **(1)** for SEC without resource-sweeping is 18.4% and 60.6% slower compared to setting **(2)** and **(6)**, respectively; vs. 4.23% and 54.9% slower for these easier model-checking benchmarks.



(a) #Refinements (log) (b) #Refinements (c) #Refinements vs k

Fig. 6. Deep simulation BMC for logic optimization on industrial (○), and public QUIP [20] and ITC99 [21] benchmarks (▽)

C. Lazy deep simulation

In *lossy* redundancy removal frameworks using only shallow fixed-depth induction [5, 16], deep simulation is unnecessary: incorrect miters are discarded along with accurate non-inductive miters. *Exhaustive* redundancy removal requires equivalence-classes to be corrected before miters can be proven. Deep simulation may arise due to deep counterexamples found by BMC or TBV, or simulating beyond shallower trace length to propagate refinements through fanout logic. While deep simulation is expensive, its refinements offset the expense of explicitly computing long miter counterexamples as NP- or PSPACE-complete problems.

Simulation consumes nearly identical runtime per timestep. When balancing solver vs. *imminent* simulation runtime considering *prior* simulation runtime, depth should be considered. We introduce the concept of *lazy deep simulation* using its own *deep-simulation trace storage*, distinct from traditional *eager shallow simulation* for minimal-depth simulation. *Eager shallow simulation* is faster, and should occur more-frequently using fewer traces. Very-shallow simulation may be parallelized via simulating slightly-overlapping windowed components [5, 16, 17]. Lazy deep-simulation is slower, evaluating the sequential netlist. It thus should occur less-frequently, sometimes accumulating more traces into *multiple* machine words to leverage multi-word simulation speedup [19].

Shallow vs. deep simulation benefit from different resource-balanced parameters affecting how many timesteps beyond trace length to simulate: a *maximum extension* parameter, and an *inactivity limit* to early-terminate the extension if insufficient refinements occur during the prior n timesteps.

We evaluate lazy deep simulation during BMC in Fig. 6 for industrial and public logic optimization benchmarks. Deep simulation is run when 640 patterns are accumulated, with a maximum overall 900-second timeout early-terminated with maximum extension of 2048 (vs. 10 for shallow), and inactivity-limit of 100 (vs. 4 for shallow). Deep simulation enables a median 15.78% more refinements (Fig. 6a), and median 30.15% compared to disabled sequential resource-sweeping (Fig. 6b). Fig. 6c shows per-k refinement for the deep benchmark of Fig. 4c. Vertical lines indicate running deep simulation, thrice after accumulating 640 patterns and once as

BMC terminates. Each deep simulation provides 12.56% more refinements on average, improving BMC (and subsequent proof) scalability. BMC completes 6 additional timesteps via this speedup, yielding 39.48% additional refinements.

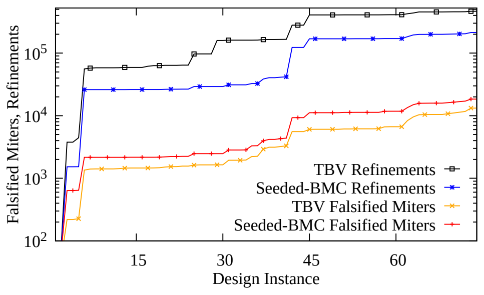
D. Seeded BMC

When a miter is falsified, simulating its counterexample often refines additional miters. Randomizing and permuting bit-parallel patterns, and simulating beyond trace length, increase the number of secondary refinements. Though the return on runtime investment, and probability of secondary refinements, via additional simulation often quickly saturates.

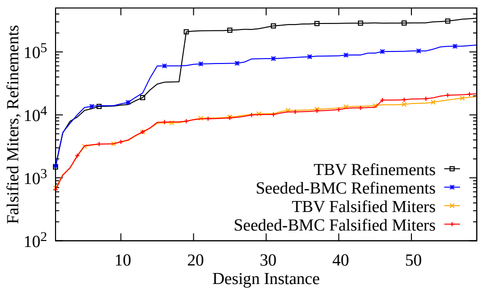
Secondary refinements may be obtained via semi-formal methods: when simulation encounters an interesting state (e.g. one that refines an equivalence-class), one can seed BMC into that state, trying to falsify additional miters [19]. If miters are falsified by seeded BMC, eager simulation of those counterexamples can be accelerated starting from the seeded state vs. initial state; especially valuable for very-deep TBV traces. Seeded-BMC traces may also be appended to the *deep-simulation trace storage* to increase the probability of secondary refinements during later deep simulation. To balance overall runtime, seeded BMC runtime may be configured to a fraction of time-elapsed to obtain those counterexamples.

One challenge to maximizing seeded BMC refinements is that controllability to propagate the new refinement scenario into adjacent logic may be limited by prior input valuations locked into the seeded state. It thus is often useful to seed BMC into a state several timesteps *before* the refinement of interest. The optimal number of timesteps varies by netlist and by miter. It can be approximated by the number of registers along simple paths between the refined equivalence class and primary inputs. It can also be varied, dynamically adjusting to a setting suitable for the netlist.

We evaluate seeded BMC in Fig. 7a during logic-optimization of public design benchmarks. Seeded BMC is run *directly after* refining TBV counterexamples, starting from already-refined equivalence-classes; their refinements are thus complementary. Seeded BMC runtime is limited to the lesser of 200 seconds, 25% of TBV runtime to produce the trace, or 50 timesteps without refinement. Seeded BMC thus consumes at most 25% of TBV runtime; in these experiments it consumed less than 3%. Despite the advantages given to TBV, seeded BMC yields 39.2% more falsified miters than TBV, while TBV generates 2.18× as many refinements. Fig. 7b identically evaluates larger proprietary industrial designs. Seeded BMC yields 11.1% more falsified miters than TBV, while TBV generates 1.7× as many refinements.



(a) QUIP [20] and ITC99 [21] benchmarks



(b) Industrial benchmarks

Fig. 7. Cumulative seeded BMC vs. TBV refinements for logic optimization

E. Approximating pathologically-deep logic

Industrial netlists often comprise sequentially-deep logic components, such as large counters or linear-feedback shift-registers (LFSRs). Incorrect miters dependent upon such components might be falsifiable with traces of minimum-length exponential with respect to their register count. Such components pose two challenges. **(i)** BMC and induction become unscalable too-shallowly to converge very-deep miters. TBV is often necessary, using heavier model-checking algorithms to solve PSPACE-complete miters, including localization [22] and logic reductions to reduce miter size, phase abstraction [23] to reduce sequential depth, well-tuned IC3 [24, 25] and BDD-based reachability [26] as general solvers, and semi-formal bug-hunting [19]. **(ii)** Simulating a very-deep trace may be prohibitively slow. While TBV may use tailored techniques to analyze deep miters (e.g. parameterized traces [27]), speculative-reduction and logic transformations applied within TBV require simulating traces on the original netlist (not only its deep components) to enable precise refinements. For extremely-deep traces, potentially billions of timesteps long, lossy shortcuts are inevitable. The following are useful minimally-lossy strategies.

(1). As refinement depth or BMC bound exceed a configurable threshold, a *depth-compensation graph* can annotate gates with: **(i)** most-recent refinement depth; **(ii)** most-recent bounded proof depth; **(iii)** the snapshotted equivalence-class for which the former were obtained. The behavior of known-deep gates can be randomly permuted when simulating bit-parallel *copies* of a trace. Because the trace pattern itself is not permuted, precise refinements occur. The random-permutation may cause pessimistic refinements, discarding valid miters. To limit the pessimism, permutation can be synchronized across the snapshotted equivalence-class gates, similar to induction counterexamples [28]. If new refinements occur, simulation may be rolled-back, and newly-mismatched candidates may be permuted for continued simulation. This process can flatten a pathological sequence of double-length counterexamples across refinements to near-linear simulation depth, similar to forcing X-pessimism to accelerate convergence in three-valued approximate reachability analysis [29].

(2). Seeded BMC (Sec. III-D) can falsify deeper miters from a simulated state, though often becomes unscalable too shallowly to converge on large counters. It is sometimes useful to *overapproximate* seeded BMC, randomly permuting the value of known-deep gates, to be less dependent upon simulation probabilities to propagate permuted values.

(3). In cases, the first concerningly-deep counterexample is too deep to simulate, or even to generate. E.g., if a design has a 64-bit LFSR, each bit may toggle shallowly, though a compare-to value may only be reached incredibly-deeply. If simulating a counterexample is impractical, the offending miter may be blindly refined as if unsolved, losing the ability to propagate refinements to fanout logic. Simulation permutation of deep gates may nonetheless be useful in other ways, e.g. reusing previous (or future) counterexamples [14].

F. Linear Proof Graph construction

The Proof Graph [10] has one node per miter. Fanout edges represent the miters whose behavior is compromised by speculative-reduction until that node's miter is proven. The miters correlating to a node may be merged as soon as its fanin miters are proven, regardless of other unproven miters. Early merging has many runtime benefits (Sec. II-A3).

The Proof Graph is reconstructed whenever a new speculatively-reduced netlist is created, at each iteration of Fig. 1. Scalable construction is thus critical. *Lossy* redundancy removal, e.g. using shallow fixed-depth induction [5], may choose to not use a Proof Graph, instead deferring all merging until fixed-point. Despite the overhead of deferred merging, e.g. causing repeated proving of accurate miters across refinements, this shortcut is motivated by the traditional runtime overhead of constructing the Proof Graph vs. lossy-solver runtime. Early merging is a practical necessity to enable *exhaustive* redundancy removal on large netlists, which requires solving PSPACE-complete non-inductive miters via general model-checking algorithms. In this section, we describe a scalable graph-labeling algorithm to compute a minimally-sized Proof Graph.

When using arbitrary model-checking algorithms to solve miters, *all* transitive fanin dependencies are recorded in the Proof Graph. Traditional iterative construction (e.g. [10] Alg. 7) traverses the fanin of each miter m' to find the speculatively-reduced gates M' which affect its behavior (stopping at vs. recursing through M', to contain runtime); edges from M' to m' are iteratively added to the Proof Graph. This initial Proof Graph can be vastly larger than a condensed version due to duplicate and transitively-implied edges, risking memout. Postprocessing is proposed to *condense* the Proof Graph to be irredundant and acyclic [10], reclaiming memory before TBV. Though iterative fanin traversal and compaction are often a runtime bottleneck on large netlists.

We propose a method to directly compute an optimally-sized Proof Graph (Fig. 8), using a single netlist traversal. Our algorithm uses an efficient graph-labeling approach [31, 32],

```
createProofGraph_graphLabeling (Spec-Reduced Netlist $N'$, Miters $M$)
 1: Compute SCC within $N'$  # Tarjan's linear algorithm [30]
 2: for each gate $g \in$ topologically-sorted gates in $N'$ :
 3:   if $g$ is not in a multi-gate SCC :
 4:     if $g$ is speculatively-reduced :  # $g$ is the non-rep. gate of a miter
 5:       Miter $m$ := miter corresponding to $g$
 6:       # add dependencies of $m$ in the fanin of $g$
 7:       for each miter $n$ with index i such that bitvector($g$)[i] $\equiv$ 1 :
 8:         add_edge($n$, $m$)  # add dependency $n \to m$ to graph
 9:       # create singleton bitvector for miter $m$
10:       unsigned idx := get unique index for miter $m$
11:       clear bitvector($g$); bitvector($g$)[idx] := 1  # singleton bitvector
12:     # copy / union bitvector to fanout gates
13:     for each gate $h$ in the fanout of $g$ :  # propagate to fanout
14:       if $h$ is part of SCC $S$ : $h$ := representative gate of SCC $S$
15:       bitvector($h$) := bitvector($h$) $\cup$ bitvector($g$)  # copy / union
16:     delete bitvector($g$)  # cleanup
17:   else if $g$ is representative gate of SCC $S$ :
18:     if $S$ contains miters :
19:       # add cyclic dependencies between miters in SCC $S$
20:       Miters $M$[ ] := get all miters in $S$, unsigned j := 0
21:       while j+1 < size($M$) :
22:         Miter $n$ := $M$[j], Miter $m$ := $M$[j+1]
23:         add_edge($n$, $m$)  # add dependency $n \to m$ to graph
24:       Miter $n$ := $M$[size($M$)], Miter $m$ := $M$[0]
25:       add_edge($n$, $m$)  # add dependency $n \to m$ to graph
26:       # add dependencies for miters in the fanin of SCC $S$
27:       Miter $m$ := miter corresponding to representative gate of SCC $S$
28:       for each miter $n$ with index i such that bitvector($g$) [i] $\equiv$ 1 :
29:         add_edge($n$, $m$)  # add dependency $n \to m$ to graph
30:       # create singleton bitvector for miter $m$
31:       unsigned idx := get unique index for miter $m$
32:       clear bitvector($g$); bitvector($g$) [idx] := 1  # singleton bitvector
33:     # copy / union bitvector to fanout gates
34:     for each gate $h$ in the fanout of gates in $S$ :  # traverse to fanout
35:       if $h$ is part of SCC $T$ : $h$ := representative gate of SCC $T$
36:       bitvector($h$) := bitvector($h$) $\cup$ bitvector($g$)
37:     delete bitvector($g$)  # cleanup
```

Fig. 8. Graph labeling algorithm for Proof Graph construction.

propagating miter-dependency information as a *bitvector*. Each speculatively-reduced gate is represented with a unique bit-index, though all miters within an SCC reuse a single bit-index, yielding a massive practical compaction.

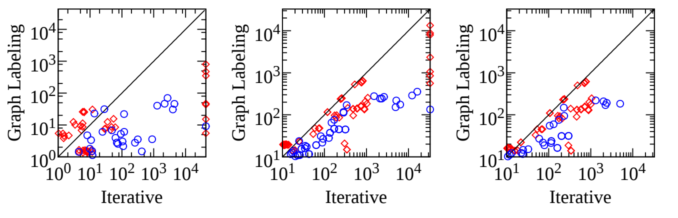
Linear SCC identification [30] identifies the strongly-connected components; each gate within an SCC is given the bit-index of a *representative* miter therein (line 1). The algorithm then iterates gates in a topological order, propagating fully-populated bitvectors denoting miter dependencies to fanout logic. When gate g is traversed, bitvector copy or union operators propagate g's bitvector to fanout gates (lines 13–15 and 34–36), accumulating their fanin dependencies. If g is a speculatively-reduced gate, all miters with indexed bits set to 1 in its bitvector are added as Proof Graph fanin edges to the node for g's miter (lines 7–8). Fanout edges of speculatively-reduced gates propagate only that gate's corresponding bit-index vs. all transitive fanin dependencies (lines 10–11).

A single bitvector is maintained for all nodes in an SCC, at its representative gate g (line 14, 35). If the SCC contains miters, a single unique bitvector index for a representative miter m therein is associated with all of that SCC's miters (line 27). The dependencies (asserted bitvector indices) of all SCC inputs become Proof Graph fanin edges of the SCC-

TABLE I
PROOF GRAPH CONSTRUCTION RESOURCES FOR LOGIC OPTIMIZATION

#	#Gates	#Miters	Iterative [10]		Graph Labeling (Fig. 8)	
			Time (s)	Memory	Time (s)	Memory
`b1`	4,476,762	192,445	4368.8*	**340 GB***	9.12	134.5 MB
`b2`	1,043,224	969,135	889.3	15.8 GB	3.58	353.8 MB
`b3`	833,584	517,979	420.1	11.9 GB	1.49	188.6 MB
`6s125`	3,232,742	2,885,658	>684	>**32 GB**	15.13	1.03 GB
`6s350`	3,774,149	1,511,759	>14400	5.6 GB	47.02	559.78 MB

BOLD: 32GB memout. [*]: runtime, memory estimate for complete construction.



(a) Time (seconds) (b) Peak memory (MB) (c) Graph size (MB)

Fig. 9. Proof Graph construction resources for logic optimization on industrial logic optimization (○) and model-checking benchmarks [33] (◇)

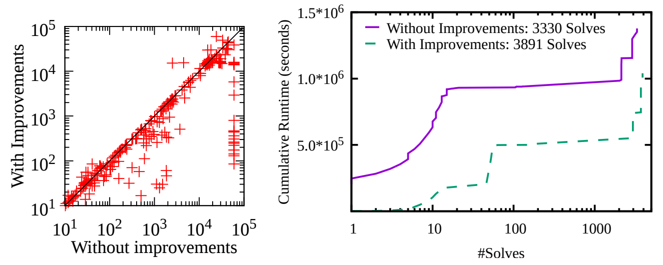
representative miter m (lines 28–29). The one-hot bitvector for m is then propagated to all SCC outputs (lines 31–32). Miters inside an SCC are added to the Proof Graph in lines 20–25 as a cyclic chain through representative m, to minimally record their inter-dependency.

Propagating a one-hot bitvector to the fanout of speculatively-reduced gates ensures a minimally-sized Proof Graph without unnecessary transitively-implied edges. When later annotating the Proof Graph with solver results, transitive traversal is generally necessary anyway [10]. Adding transitively-implied edges tends to degrade performance, bloating graph size and requiring more unsuccessful fanout-edge traversals: until a miter is proven, soundly-proven results for fanin miters cannot propagate through that miter anyway.

Theorem 1 (Sound)**.** *Given speculatively-reduced netlist N' and miters M, Fig. 8 generates a* sound *Proof Graph: every fanout dependency of speculatively-reduced gate g is transitively reachable via fanout traversal of g's Proof Graph node.*

Theorem 2 (Optimal)**.** *Proof Graph $P = \langle M, E\rangle$ generated by Fig. 8 has minimal size. I.e., there does not exist a Proof Graph P' with fewer nodes or edges correctly representing the dependencies of N' and M.*

Table I shows resources to construct the Proof Graph for selected large industrial and model-checking benchmarks, using the traditional iterative approach [10] vs. graph-labeling (Fig. 8). This highlights the scalability limitation of the former, precluding TBV on the largest netlists. Fig. 9 shows resources across more industrial and model-checking benchmarks; each run with 32GB memory-limit and 14400-second timeout. Graph-labeling is up to 98.34× (Fig. 9a) faster with a 53.77% lower peak memory requirement on average (Fig. 9b), and enables the Proof Graph creation for 17 more benchmarks within resource-limits. While the iterative approach postprocesses the Proof Graph by SCC compaction and pruning transitively-implied edges, graph labeling generates a 59.25% smaller Proof Graph without costly postprocessing (Fig.9c).



(a) SEC runtime (seconds) (b) Runtime vs # solves, model-checking

Fig. 10. Runtime for SEC and model-checking [33] benchmarks

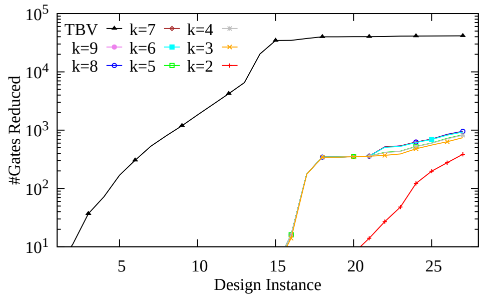
IV. EXPERIMENTAL RESULTS

In this section, we show end-to-end results for various redundancy removal applications, using the contributions of Sec. III. Though to minimize memouts, these all benefit from the scalable Proof Graph presented in Sec. III-F. Our techniques are implemented within RuleBase: Sixthsense Edition [34], building upon the techniques presented in [13, 32].

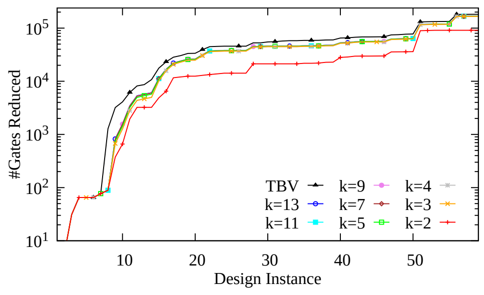
1) Property- and Equivalence-Checking: Fig. 10a compares end-to-end runtime with *vs.* without our improvements on 384 SEC benchmarks, using the refinement phase of Fig. 6 with 15-second seeded BMC, 4-hour induction, and 12-hour overall time-limits using 1-process TBV. Our improvements solve 362 common benchmarks 2.1× faster on average, with 22 unique solves. Fig. 10b shows model-checking speedup, using a similar configuration though without corresponded signal-name filtering. Cumulative runtime is 32.4% faster, and 16.9% more properties are solved using our improvements.

2) Logic Optimization: We present logic optimization results in Fig. 11a for public design benchmarks, and Fig. 11b for industrial designs. *Exhaustive* logic optimization is a global concern [35], and particularly challenging: **(1)** Unlike SEC, name-correlation cannot fragment equivalence-classes to pairs. Large equivalence-classes require *many* refinements to correct, iteratively comparing a gate to *many* different representatives. **(2)** Redundancy removal is not early-terminated when properties are solved. These benchmarks were preprocessed using 1-induction. On public benchmarks, preprocessing removes 1442832 gates. Deeper-k induction removes an additional 959 gates, then TBV an additional 40494, for 2.9% greater redundancy removal overall. Redundancy-removal was *exhaustive* for 65 of 74 netlists (containing up to 373338 AND gates and 8809 registers), capping resource at 64GB 3-day 5-process. No-reduction points are not plotted. Miters of depth ≥ 32768 (up to 250125) were encountered on 3 netlists; synchronized random-permutation (Sec. III-E) losslessly aided convergence.

The industrial benchmarks are evolving components, already redundancy-removed at prior logic iterations. Leveraging our contributions, deeper-k induction removes an ad-



(a) QUIP [20] and ITC99 [21] benchmarks



(b) Industrial benchmarks

Fig. 11. Cumulative deep-k induction and TBV logic optimization

ditional 167462 gates (76259 with $k \geq 3$), and TBV an additional 14248. Practical benefits were significantly greater due to multiple on-chip copies of each component. Exhaustiveness was achieved for 11 of 59 netlists (containing up to 857110 AND gates and 75952 registers). Miters of depth ≥ 32768 (some above 100M) were encountered on 21 netlists. Without the contributions presented in this paper, exhaustive redundancy removal was achievable for only 2 netlists.

V. Conclusions

Global energy use of semiconductor devices doubles every three years, mandating a new type of Moore's Law for energy efficiency [35]. Sequential redundancy removal is one of many remedies, and has many other applications.

We introduce various techniques to improve the scalability of sequential redundancy removal, eliminating bottlenecks to *exhaustiveness*. *Sequential resource-sweeping* self-tailors to netlist depth, yielding $> 20\%$ greater equivalence-class refinement via BMC and $\approx 5\%$ greater inductive redundancy removal, on average. *Resource-balanced simulation* accelerates redundancy-removal by $\approx 30\%$ on average. *Lazy deep simulation* yields $\approx 16\%$, and *seeded-state BMC* $\approx 34\%$, additional refinements with minor runtime overhead. Graph-labeling *Proof Graph* construction boosts scalability by two orders of magnitude, making TBV practical on very-large netlists. Heuristics are introduced to prevent pathologically-deep logic from derailing convergence. Overall, our techniques yield 2.1× speedup to SEC, 32.4% speedup with 16.9% more solves on large difficult model-checking problems, and enabled *exhaustive* redundancy removal on large netlists up to 857110 AND gates, 75952 registers.

Acknowledgments

We would like to thank the reviewers for their valuable feedback and comments. Kristin Y. Rozier is supported by NSF:CCRI grant 2016592 and NSF:CAREER grant 1664356.

References

[1] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 409–426, Oct 2000.

[2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Design Automation Conference (DAC)*, pp. 463–466, Jun 2005.

[3] R. Brayton, N. Een, and A. Mishchenko, "Using speculation for sequential equivalence checking," in *International Workshop on Logic and Synthesis (IWLS)*, Jun 2012.

[4] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *Design, Automation and Test in Europe (DATE)*, pp. 618–623, Feb 1998.

[5] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, 2008.

[6] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "DeltaSyn: An efficient logic difference optimizer for ECO synthesis," in *2009 International Conference on Computer-Aided Design (ICCAD)*, pp. 789–796, 2009.

[7] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *International Conference on Computer-Aided Design (ICCAD)*, November 2004.

[8] D. Brand, "Verification of large synthesized designs," in *International Conference on Computer-Aided Design (ICCAD)*, November 1993.

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, 2002.

[10] M. Case, J. Baumgartner, H. Mony, and R. Kanzelman, "Optimal redundancy removal without fixedpoint computation," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 101–108, Oct 2011.

[11] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems* (W. R. Cleaveland, ed.), (Berlin, Heidelberg), pp. 193–207, Springer Berlin Heidelberg, 1999.

[12] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 127–144, Nov 2000.

[13] R. Dureja, J. Baumgartner, R. Kanzelman, M. Williams, and K. Y. Rozier, "Accelerating parallel verification via complementary property partitioning and strategy exploration," in *Formal Methods in Computer-Aided Design (FMCAD)*, Sep 2020.

[14] S. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. D. Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2022.

[15] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 836–843, November 2006.

[16] A. Mishchenko and R. Brayton, "Integrating an AIG Package, Simulator, and SAT Solver," in *International Workshop on Logic and Synthesis (IWLS)*, June 2018.

[17] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis, "Parallel combinational equivalence checking," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Oct 2019.

[18] "Hardware Model Checking Competition 2017, Single property benchmarks," *http://fmv.jku.at/hwmcc17*.

[19] P. K. Nalla, R. K. Gajavelly, J. Baumgartner, H. Mony, R. Kanzelman, and A. Ivrii, "The art of semi-formal bug hunting," in *International Conference on Computer-Aided Design (ICCAD)*, ACM, 2016.

[20] "Altera corporation/intel: Quartus II University Interface Program," *https://github.com/ispras/hdl-benchmarks/tree/master/hdl/quip*.

[21] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Design & Test of Computers, https://github.com/cad-polito-it/I99T*, vol. 17, no. 3, 2000.

[22] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 33–51, 2002.

[23] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 1076–1082, November 2005.

[24] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, p. 70–87, 2011.

[25] R. Dureja, A. Gurfinkel, A. Ivrii, and Y. Vizel, "IC3 with Interal Signals," in *Formal Methods in Computer-Aided Design (FMCAD)*, (New Haven, CT, USA), IEEE/ACM, Oct. 2021.

[26] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 401–424, April 1994.

[27] C. Wang, A. Gupta, and F. Ivancic, "Induction in CEGAR for detecting counterexamples," in *Formal Methods in Computer Aided Design (FMCAD)*, pp. 77–84, 2007.

[28] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *Design, Automation and Test in Europe (DATE)*, pp. 1674–1679, Apr 2009.

[29] M. L. Case, J. Baumgartner, H. Mony, and R. Kanzelman, "Approximate reachability with combined symbolic and ternary simulation," in *Formal Methods in Computer-Aided Design (FMCAD'11)*, pp. 109–115, 2011.

[30] R. Tarjan, "Depth first search and linear graph algorithms," in *SIAM Journal on Computing*, 1972.

[31] G. Cabodi, P. Camurati, and S. Quer, "A graph-labeling approach for efficient cone-of-influence computation in model-checking problems with multiple properties," *Software: Practice and Experience*, vol. 46, no. 4, pp. 493–511, 2016.

[32] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier, "Boosting verification scalability via structural grouping and semantic partitioning of properties," in *Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–9, Oct 2019.

[33] "Hardware Model Checking Competition 2010-2017," *http://fmv.jku.at*, Selected all non-liveness benchmarks, filtered to largest file sizes. Fig. 10b prunes timeout-vs-timeout.

[34] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 159–173, 2004.

[35] M. Mann and V. Putsche, "Semiconductor: Supply chain deep dive assessment," February 24, 2022. United States. https://www.osti.gov/biblio/1871585 *"The global energy use of products featuring semiconductors has doubled every three years since 2010 primarily due to the accelerating use of semiconductors in all facets of our modern economy and the deceleration of energy efficiency increases due to miniaturization. This exponential growth in energy use is projected to accelerate..."*.

DAG-Based Compositional Approaches for LTLf to DFA Conversions

Suguman Bansal
Georgia Institute of Technology, USA
suguman@gatech.edu

Yash Kankariya
Georgia Institute of Technology, USA
ykankariya3@gatech.edu

Yong Li
University of Liverpool, UK
yong.li3@liverpool.ac.uk

***Abstract*—Scalable and efficient conversions of LTL *over finite horizon* (LTLf) to their *deterministic finite automata* (DFA) remain a critical bottleneck in several applications of LTLf. Recently, *compositional approaches* have seen remarkable success in scaling the conversion to large formulas. Here the input formula is first decomposed into smaller subformulas, each of which can be easily converted to their DFAs, then these DFAs are composed to generate the desired DFA. This work proposes a series of simple-yet-effective optimizations to improve the performance of compositional approaches based on reducing the number of composition steps required to generate the desired DFA.**

We incorporate these optimizations in a tool called Lisa2 that builds on one of the state-of-the-art tools for LTLf-to-DFA conversion. A comprehensive empirical evaluation of Lisa2 demonstrates overall improvements on both parameters of the number of benchmarks solved and runtime. Most remarkably, it demonstrates significant improvement over structured benchmarks where runtime speedups range between 1.5x to 8000x under fair comparisons to prior state-of-the-art tools.

I. Introduction

Linear Temporal Logic over finite traces [1] (LTLf) is the finite-horizon counterpart of Linear Temporal Logic (LTL) over infinite traces [2]. LTLf is rapidly gaining popularity among real-world applications where behaviors are better expressed over a finite but unbounded horizon. These include applications in planning and synthesis [3], [4], [5], [6], reinforcement learning [7], [8], [9], business processes [10], verification [11].

A critical challenge facing its applications is the conversion of LTLf specifications into their equivalent *deterministic finite automata (DFAs)*. This is not unexpected since the LTLf-to-DFA conversion exhibits a double-exponential blow-up in the size of the input specification in the worst-case [1], [12]. Yet, state-of-the-art LTLf-to-DFA conversion tools like Lisa [13] and Lydia [14] often succeed at converting large formulas. Their success can be attributed to *compositional algorithms* which are split into two phases. First, in the *decomposition phase* a large LTLf formula is decomposed into smaller subformulas using the formula's *Abstract Syntax Tree* (AST). Next, in the *composition phase*, subformulas at the leaves of the AST are converted to their DFAs using direct LTLf-to-DFA tools suitable for smaller formulas, such as Spot [15] or Mona [16]. Then, these DFAs are composed using language-theoretic and/or automata-based operations to obtain the DFA of the original formula. For DFA composition, the AST of the formula is traversed bottom-up.

Through this work, we propose a series of simple yet effective optimizations to improve the performance of compositional algorithms. Our first optimization aims at striking a balance between the time spent in converting subformulas at the leaves of the AST into their DFAs and the time spent in composing the intermediate DFAs. The deeper the AST is unrolled, the smaller are the subformulas at the leaves of the AST. While these smaller subformulas may be easier to convert into their DFAs, it increases the number of composition steps required to traverse the AST. To this end, we propose to unroll the AST on their outermost boolean operators only. In contrast, both Lisa and Lydia unroll at the extremes: Lisa unrolls on the outermost conjunction only whereas Lydia unrolls completely till the propositional literals.

Our other optimizations focus on reducing the number of composition steps required during the bottom-up traversal by modifying the AST. Our first optimization is based on eliminating subformula duplication within the AST. This eliminates multiple computations of the DFA of the same subformula that may be present at multiple locations in the AST of a formula. Such duplication is not uncommon in structured, real-world formulas. Our second optimization is based on using semantics-preserving syntactic transformations to the formula. In particular, subformulas of the form $\phi = \bigwedge_{i=1}^{k}(\psi \vee \phi_i)$ are rewritten as $\phi = \psi \vee \bigwedge_{i=1}^{k} \phi_i$, as the latter requires fewer composition steps: The former representation of ϕ will require $2k-1$ composition steps (k steps to create the DFA for all $(\psi \vee \phi_i)$ and $k-1$ steps from the outer conjunction of these clauses). Whereas, the latter will require only k steps of which $k-1$ steps are required to create the DFA for the large conjunction and one more is required to compose with ψ. Neither Lisa nor Lydia incorporate either of these optimizations.

We have implemented these optimizations in a tool Lisa2 that builds on the existing tool Lisa. The compositional algorithm in Lisa2 differs from Lisa as follows: (a). In the decomposition phase, the formula is unrolled on all outermost boolean operations using the formula's AST, (b). Next, there is an additional *optimization phase* in which duplicate removal and syntactic transformations modify the AST to a DAG as opposed to the AST, (c). Finally, in the composition phase, formulas at the *leaves* of the DAG are converted to their DFA and then these intermediate DFAs are composed in a

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_28

bottom-up traversal of the DAG. Lisa2 also differs from Lisa and Lydia in supporting multiple representations for DFAs. It permits Spot's [15] labeled-graph and Reduced-Ordered BDD (ROBDD) [17] (which is used by Lisa) as well as Mona's Shared Multi-Terminal BDD (ShMTBDD) (which is used by Lydia) [16]. Permitting both datastructures makes Lisa2 more versatile than prior tools. An additional benefit is that this enables fair comparison with Lisa and Lydia. While these prior tools implement differing compositional approaches, a fair comparison of these algorithms has not been possible since the performance of these tools is also affected by the complementary strengths of the underlying datastructure for DFAs. In particular, ROBDD may be slower but require less memory whereas ShMTBDD can be blazingly fast but are memory exhaustive. With the flexibility in choice of DFA datastructure, Lisa2 can compare different algorithmic approaches by ensuring that their underlying datastructures are identical, hence xrendering fairer comparisons.

A comprehensive empirical evaluation demonstrates significant improvements over prior state-of-the-art tools in both the number of benchmarks solved and their runtime. We evaluated the performance of Lisa2 against Lisa and Lydia on LTLf benchmarks (a collection of randomly generated formulas and structured formulas) from the LTLf track in SYNTCOMP2023[1]. While Lisa2 outperforms both baselines comprehensively, its performance on the structured benchmarks is most remarkable. Not only Lisa2 solves $\sim$50% more structured benchmarks than prior approaches, it also demonstrates runtime improvements in the range of 1.5x-8000x (with more benchmarks recording high runtime improvement), highlighting the strength of our tool on realistic benchmarks.

II. Preliminaries and Notations

A. Linear Temporal Logic over Finite Traces (LTLf)

LTLf [1] extends propositional logic with finite-horizon temporal operators. The syntax of LTLf over a finite set of propositions Prop is identical to LTL, and defined as

$$\varphi := \mathsf{true} \mid \mathsf{false} \mid a \in \mathsf{Prop} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{X}\varphi \mid \varphi_1 \mathrm{U} \varphi_2$$

where X (Next) and U (Until), are temporal operators. We include their dual operators, N (Weak Next) and R (Release), defined as $\mathrm{N}\varphi \equiv \neg\mathrm{X}\neg\varphi$ and $\varphi_1\mathrm{R}\varphi_2 \equiv \neg(\neg\varphi_1\mathrm{U}\neg\varphi_2)$. We also use typical abbreviations such as $\mathrm{F}\varphi \equiv \mathsf{true}\mathrm{U}\varphi$, $\mathrm{G}\varphi \equiv \mathsf{false}\mathrm{R}\varphi$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$. The semantics of LTLf can be found in [1].

Wlog, we assume formulas are given in *negation normal form* (NNF), i.e., the negation operator ($\neg$) appears in front of propositions only. In the given syntax, all formulas can be converted to their NNF with no blow-up in length.

Every LTLf formula φ over Prop can be converted into a deterministic finite-state autoamta (DFA) D with alphabet $\Sigma = 2^{\mathsf{Prop}}$ and at most double-exponential number of states in $|\varphi|$ such that $\mathcal{L}(D) = \mathcal{L}(\varphi)$ [1].

[1]https://www.syntcomp.org

B. Abstract Syntax Tree

The *Abstract Syntax Tree (AST)* is an n-ary tree representation of an LTLf formula. Formally, for an LTLf formula ϕ, (1). Every node corresponds to a subformula of the formula φ. In particular, the root of the AST corresponds to the formula itself, (2). For every node, the *operator* of the node is given by the outermost (primary) operator of its subformula, (3). For every node, its children correspond to the immediate subformulas of the formula in that node. Formulas with unary operators (such as $\neg$, X, F, and so on), binary operators (U and R), and n-ary operators ($\vee$ and $\wedge$) consist of one, two, and n-many children, respectively. The order of children is crucial for temporal operators U and R. For the remaining operators, the order of children does not alter the semantics of the formula since the operators are commutative.

III. Related Work

Optimizations in compositional LTLf*-to-DFA approaches.:* The current state-of-the-art AST-based compositional tools, Lisa [13] and Lydia [14], employ various optimizations to counter its inherent non-elementary complexity. In addition to aggressive DFA minimization [16] and order in which DFAs at each node are composed [13], the tools optimize on the depth to which the ASTs are unrolled during formula decomposition. For instance, Lisa decomposes the original formula at its outermost conjunction only, thus creating a k-ary AST of depth one. Thus, the leaves represent subformulas as opposed to atomic propositions. On the other hand, Lydia [14] generates the full k-ary AST up to literals in the leaves. This way tools attempt to trade-off between resources spent in the direct conversion of subformulas at the leaves and composition steps. The tools also optimize on the data structure used for explicit state-space representations of the DFAs. Lisa stores DFAs as labeled-graphs and in symbolical state-space using *Reduced-Ordered Binary Decision Diagrams (ROBDD)* [17] if necessary. Lydia stores DFAs using *Shared Multi-Terminal Binary Decision Diagrams (ShMTBDD)* of Mona [16].

Direct LTLf*-to-DFA conversions.:* Spot [15] and Mona [16] are two popularly used tools for direct LTLf-to-DFA conversions of smaller formulas. Spot translates the LTLf formula into an LTL formula with equivalent semantics, converts this formula into a Büchi automaton [18], and then transforms this Büchi automaton into a DFA. The Mona-based approach translates LTLf into *first-order logic over finite traces (FOL)* and then Mona converts the FOL formula into a DFA. Both generate minimal DFAs in explicit state-space representation.

DAG-based compositional approaches.: Mona also uses directed acyclic graphs (DAGs) to represent formulas as we do in this work [19]. The differences lie in the ways we identify equivalent subformulas and the depth of the DAGs. Mona identifies equivalent formulas ϕ and ϕ' by checking whether there is an order-preserving renaming of propositions in ϕ such that ϕ and ϕ' are identical, while we check the syntax equivalence of two formulas, simpler but much more efficient. Moreover, Mona's DAG unrolls till its literals while our DAG unrolls on boolean operators only. To the best of our knowledge, Lisa and Lydia do not use DAG since no intermediate DFAs are stored for later use in their source code.

Optimizations in LTL to automata conversion: The conversion of LTL (Linear Temporal Logic) [2] to automata forms has received much attention. While the conversion incurs a single-exponential and double-exponential blow-up for the non-deterministic and deterministic versions automata versions, significant work has gone into developing optimizations for LTL to automata. However, these optimizations are too sophisticated for LTLf to automata translations, where relatively simpler translations have been shown to be more effective.

IV. Optimizations

We propose a series of optimizations to be applied to compositional approaches for LTLf-to-DFA conversion. The first optimization (Section IV-A) determines the depth to which an input formula's AST is unrolled during formula decomposition into smaller subformulas. The remaining two optimizations are geared to reduce the computation required during the composition phase by reducing the number of composition operations. Here, the first optimization compresses this AST by removing duplicate subformulas (Section IV-B). The second optimization performs semantics-preserving syntactic transformations that are guaranteed to reduce the number of composition steps (Section IV-C).

A. Depth of AST Unrolling

Our first optimization is based on the depth to which an input LTLf formula is unrolled to obtain smaller subformulas. We propose to unroll subformulas only if their outermost operator is a boolean operator. Recall, since we assume formulas are given in NNF, the outermost boolean operators are effectively only the conjunction operator or the disjunction operator as negations appear before propositions only. Consequently, for formulas at the leaves of this tree, the outermost operator could be any of the temporal operators. Figure 1 illustrates such an unrolling of an AST.

We make this choice to strike a balance between the conversion of subformulas to DFAs at the leaves vs. the composition of DFAs at intermediate nodes to obtain the final DFA. Prior state-of-the-art approaches Lisa and Lydia take diametrically opposite routes in this regard. Lisa unrolls the AST at its outermost conjunction only. I.e., given an LTLf formula $\varphi = \bigwedge_{i=1}^{n} \varphi_i$, Lisa decomposes the original formula into the n-subformulas given by φ_is. The disadvantage of this decomposition is that in the worst case, the φ_is could be too large to be handled by an off-the-shelf LTLf-to-DFA conversion tools like Spot or Mona. The advantage, however, is that once the DFAs for the φ_is are created, the approach requires only $n-1$ composition steps, where each composition consists of polynomial-time operations of DFA product and DFA minimization only. In contrast, Lydia unrolls the AST completely. I.e., its leaves comprise of literals (propositions or their negation). This ensures that the DFA at the leaf node is obtained trivially. However, not only do the number of composition operations increase dramatically, the composition operations may become more complex. In particular, the composition at nodes with a boolean operator comprise of polynomial-time operations identical to Lisa, but the composition at nodes with temporal operators may involve exponential-time operations such as projection and/or determinization.

Our choice to unroll only on boolean operators ensures that all composition operations require polytime operations only while also ensuring that the size of subformulas at the leaves are small, hence combining the benefits of Lisa and Lydia.

B. Duplicate Removal

For our next optimization, we observe that in several formulas, an intermediate subformula may appear more than once in the formula's AST. This results in redundant computation during the composition phase as it generates the DFA for equivalent subformulas multiple times. To eliminate such redundant recomputation, we propose to merge nodes of equivalent subformulas. This is illustrated in Figure 2 where formula θ_1 that appeared twice in Figure 1 has been merged into one node. Such duplication removal will result in the AST being converted to a DAG as the merged nodes are required to serve multiple parent nodes.

In order to merge nodes in an AST, we must check if the formulas at two or more nodes are equivalent. LTLf formula equivalence is PSPACE-complete, hence we must resort to computationally inexpensive approaches to identify formula equivalence. Tools such as Spot offer inexpensive syntactic checks to examine formula equivalence. We combine these checks with leveraging the parent-child relationship between nodes in the AST to identify equivalent formulas.

To elaborate further, first we identify formula equivalence between the leaf nodes of the AST using syntactic checks, and merge each class of equivalent formulas into one leaf node. This converts the AST into a DAG as the merged leaf nodes will now serve multiple parent nodes. Next, it is easy to see that two non-leaf nodes are equivalent if all their children nodes are identical and their operators are identical. All such formula equivalence in non-leaf nodes can be identified efficiently in a single bottom-up traversal of the DAG that simultaneously merges equivalent non-leaf nodes into one node.

Note that this procedure may fail to recognize equivalent subformulas that do not adhere to our inexpensive checks. Despite this incompleteness, we observe that sometimes it can reduce the number of nodes in the AST/DAG by 30-40% in negligible time, hence demonstrating its effectiveness.

C. Semantics-Preserving Transformation

The final optimization aims to reduce the number of composition steps using *semantics-preserving syntactic transformation* of the formula. Lemma 1 motivates our optimization:

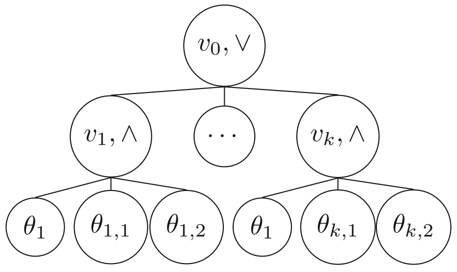


Fig. 1: AST unrolled on boolean operators only.

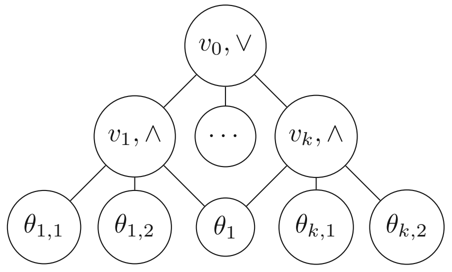


Fig. 2: After duplicate removal. θ_1 has been merged.

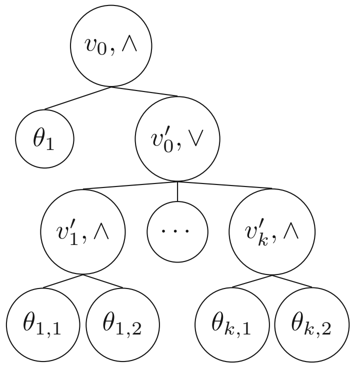


Fig. 3: After Transformation. θ_1 has been pulled out.

Lemma 1. *Consider the following formula,*

$$\phi = \circ'^{k}_{i=1}\Big((\theta_1 \circ \cdots \theta_l) \circ (\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big) \tag{1}$$

where $\circ, \circ' \in \{\vee, \wedge\}$ s.t. $\circ' \neq \circ$, and for all $i \in [k]$, $m_i \geq 0$.

Then ϕ is equivalently written as:

$$\phi' = (\theta_1 \circ \cdots \theta_l) \circ \Big(\circ'^{k}_{i=1}(\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big) \tag{2}$$

using the laws of associativity. Assuming the DFAs for all θ_i and $\theta_{i,j}$ are given, the required composition steps to create the DFA for ϕ is $\mathcal{O}(l \cdot (k-1))$ more than those required to create the DFA for ϕ'.

Proof. We begin with a sketch of the argument. In practice, a product over k DFAs requires $k-1$ products between two DFAs. Now, notice that the intermediate DFA for the common segment $\theta_c = (\theta_1 \circ \cdots \theta_l)$ is constructed $k-1$ times more in ϕ that in ϕ'. By pulling out the common segment θ_c using the laws of associativity in ϕ', the DFA for θ_c is constructed only once, amounting to the difference.

The formal argument follows: Let us first analyze the number of products required in ϕ. For every $i \in [k]$, the clause $(\theta_1 \circ \cdots \theta_l) \circ (\theta_{i,1} \circ \cdots \theta_{i,m_i})$ requires $l + m_i - 1$ products. Next, these clauses are combined using products to obtain ϕ. Since there are k clauses, we require $k-1$ additional products. Therefore, evaluating ϕ requires $(k-1) + \Sigma_{i=1}^{k}(l-1+m_i) = k \cdot l - 1 + \Sigma_{i=1}^{k} m_i$ product operations.

Next, we analyse the number of products required in ϕ'. For every $i \in [k]$, the clause $(\theta_{i,1} \circ \cdots \theta_{i,m_i})$ requires $m_i - 1$ products. In addition, these k clauses are combined using $k-1$ products to form the DFA for $\Big(\circ'^{k}_{i=1}(\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big)$. Hence, $\Big(\circ'^{k}_{i=1}(\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big)$ requires $k - 1 + \Sigma_{i=1}^{k}(m_i - 1) = (\Sigma_{i=1}^{k} m_i) - 1$ operations. Combined with l many θ_js to obtain ϕ', we require $l - 1 + \Sigma_{i=1}^{k} m_i$ products to form ϕ'. Therefore, constructing the DFA via ϕ' requires $\mathcal{O}(l \cdot (k-1))$ fewer operations than creating the same DFA via ϕ, where l is the number of subformulas common to k-many clauses in ϕ. □

Our optimization, therefore, applies the associative law to transform nodes of the form ϕ to nodes of the form ϕ' in the DAG obtained after duplicate removal. As earlier, the transformation is carried out by an analysis of the parent-child relations between nodes. A node v_0 is eligible for the transformation if the formula it represents is of the form ϕ, i.e. : (a) the outermost boolean operator should differ from the outermost boolean operator of all of its children, and (b) all its children share a common child of their own, referring to $\theta_c = (\theta_0 \circ \cdots \theta_l)$ in ϕ. In the DAG, the common child of all children is simply a common grandchild node. The common grandchildren are obtained from the intersection of all of children of v_i's for $i \geq 1$. In Figure 2, node v_0 is eligible for the transformation with a single common grandchild in θ. When eligible, the transformation *pulls out* all common segment θ_c. In the DAG, this translates to promoting all common grandchildren of v_0 to direct children of v_0 and the ealier children of v_0 are modified accordingly. Figure 2 to Figure 3 illustrate the transformation. Observe that the transformation may result in the creation of new nodes such as $v_0', \cdots, v_k'$ in Figure 3.

As earlier, these transformations are carried out in a single bottom-up traversal (reverse topological order) of the DAG starting with the leaf nodes. In instances when the transformation results in the creation of a new node (such as $v_0', v_1', \cdots v_k'$ in Figure 3), the new nodes are examined for duplicates using the earlier approach. Then the transformation is recursively applied to v_0' first and then to $v_1', \cdots v_k'$ before returning to the next node as per the reverse topological order.

V. Compositional Algorithm

For the sake of completion, we present an outline of the compositional algorithm. W.l.o.g., our algorithm receives an LTLf formula in NNF and outputs a minimal DFA for the formula. The algorithm proceeds in three phases: First is the *decomposition phase* in which the input LTLf formula is decomposed into smaller subformulas based on its AST. The AST is unrolled on boolean operators only. This is followed by the *optimization phase* in which the proposed duplication removal and semantic-transformations are applied. As a result, the AST is converted to a DAG. Finally, in the *composition phase*, the subformulas at the *leaves* of the DAG, i.e. nodes with no outgoing edges in the DAG, are converted to their minimal DFA form. Next, the DAG is traversed bottom-up starting with the leaves, i.e. the DAG is traversed in reverse topological order. During this traversal, the minimal DFA at a node is created if the minimal DFA at all its children have already been constructed. The primary difference from the AST-based composition is that the DFA at a node in the AST can be removed from memory as soon as the DFA in its parent node has been constructed. In the case of a DAG, the DFA at a node is discarded only after the DFA at *all* its parent nodes have been generated.

VI. Implementation Details

We have implemented compositional algorithm in a tool called Lisa2[2]. Lisa2 takes an LTLf formula in NNF as its input and outputs its minimal DFA in explicit representation.

In brief, Lisa2 extends a current state-of-the-art tool Lisa to incorporate the optimizations described in Section IV. In detail, Lisa2 has been written in C++. It uses Spot LTLf parser to parse the input formula. The input formula is decomposed into a DAG following the optimizations described in Section IV. To convert the subformulas at the leaves of the DAG, Lisa2 converts the LTLf formulas at the leaf nodes to their equivalent first-order logic (FOL) and uses Mona to convert the FOL formulas to their minimal DFAs. The DFAs are then composed as described in Section V. Similar to Lisa [13], Lisa2 deploys two performance-enhancing heuristics (a) *aggressive DFA minimization*, i.e. each DFA (intermediate of final) is minimized as soon as it is created, and (b) *smallest-first* heuristic that always picks the smallest two (minimal) DFAs to compose during a k-way product construction (for both conjunction and disjunction).

Lisa2 generates DFA in explicit-state representation, i.e., the states are given explicitly and the transitions are given as labeled formulas over the propositions of the input LTLf formula. Lisa2 supports two datastructures to represent the final and all intermediate DFAs: (a) Spot's labeled graphs and Reduced Ordered BDD (ROBDD) and (b). Mona's Shared Multi-Terminal BDD (ShMTBDD). We refer to these two variants of our tool as Lisa2-Spot and Lisa2-Mona, respectively. These tool variants use the DFA manipulation APIs provided by Spot and Mona, respectively, for all DFA operations including the product construction and minimization.

Tool Features: By supporting both Spot and Mona, Lisa2 is the only LTLf-to-DFA conversion tool that can support both datastructures, adding to its versatility in applications. Another motivation to support both DFA datastructures is to enable fair comparison for future algorithmic advances in LTLf-to-DFA conversion tools. Prior tools Lisa and Lydia support only one of the two Spot's labeled graphs + ROBDD combination and ShMTBDD, respectively. These datastructures have complementary benefits (ROBDD may be slower but require less memory whereas ShMTBDD are faster but are memory extensive.) and a bear significant impact the performance of their tools. As a result, performance comparisons between prior tools are unable to differentiate between the improvement caused by the algorithm vs. the improvement caused by the datastructure. Thus the ability to switch between DFA datastructures within Lisa2 creates the opportunity for more fair comparisons of algorithmic advances in LTLf-to-DFA tools.

A. Implementation-Level Optimizations

Lisa2 incorporates several implementation-level optimizations. Few key ones are described below.

First, formulas of the form $\mathrm{G}(\bigwedge_{i=0}^{m} \phi_i)$ and $\mathrm{F}(\bigvee_{i=0}^{m} \phi_i)$ are equivalently written as $\bigwedge_{i=0}^{m}(\mathrm{G}\phi_i)$ and $\bigvee_{i=0}^{m}(\mathrm{F}\phi_i)$, respectively, to promote deeper decomposition on boolean operators. Had the formulas been retained in their earlier format, then the formulas would not be decomposed any further since the outermost operator is temporal. This optimization generates smaller subformulas.

Second, Lisa2 already identifies few subformula duplications (using Spot's inexpensive methods to determine formula equivalence) during the unrolling of the formula's AST. As a result, the outcome of the unrolling may already be a DAG as opposed to an AST. We do this as we observed that in some cases, the AST obtained from unrolling on boolean operators could become very large. Combining the unrolling with a shallow duplication-removal curb the growth in the AST.

We observe that in practice most DAG/AST nodes do not possess a common grandchild, making the node ineligible for the semantics-preserving transformation. Instead, it is more likely that several nodes possess a *popular* grandchild that may be a child of most but not all children of the node. In these cases, we perform the transformation only with the children that share the popular grandchild.

VII. Experimental Analysis

A. Design and Setup for Empirical Evaluation

Baselines and Fair Comparisons: We compare Lisa2 to the three state-of-the-art baselines: Lydia, Lisa, and Lisa-Explicit. All three tools are based on compositional algorithms. They differ in the depth of unrolling, few algorithmic details, and the underlying DFA datastructure. Lydia unrolls to the literals whereas Lisa and Lisa-Explicit unroll on the outermost conjunction only. In terms of DFA data structure, Lydia uses Mona's ShMTBDD while Lisa and Lisa-Explicit use Spot's labeled-graphs and ROBDDs. Since tool performance is known to be impacted by the DFA datastructure, we establish the following fair comparisons:

- Lisa2-Mona vs. Lydia
- Lisa2-Spot vs. Lisa and Lisa-Explicit

All tools accept inputs in Spot-parsable format, ensuring consistency among tools in input format.

[2]https://github.com/suguman-lab/lisa2

Benchmarks: We use benchmarks from the LTLf-track at SYNTCOMP 2023[3]. We evaluate on 490 benchmarks of which 400 formulas are generated randomly and the remaining 90 are structured formulas derived from the "two-player games" category. Among the structured benchmarks, we use 20, 10, and 60 benchmarks from the single counter, double counter, and nim benchmark classes, respectively.

Set-up: All experiments were conducted on a single node of a high-performance cluster (https://pace.gatech.edu/). Each node consists of four quad-core Intel-Xeon processors running at 2.6 GHz with 4hrs timeout and 16GB of RAM each.

B. Performance-Related Observations

We begin by examining the performance of Lisa2 against its counterparts w.r.t. runtime and number of benchmarks solved. Overall, Lisa2 not only solves more benchmarks that all other counterparts, it also improves the runtime significantly. Most remarkable is its performance on the structured benchmarks where Lisa2 solves ∼50% more benchmarks and displays runtime improvements in the range of 2x-8000x. We describe our observations and inferences in detail below.

Lisa2 demonstrates the best overall performance: The cactus plots of the performance of all tools in Figure 4a (cactus plot on all benchmarks) and Figure 4b (cactus plot on structured benchmarks only) demonstrate that variants of Lisa2 solve the most number of benchmarks in both cases. Recall the fair comparisons from the previous section. We observe that on all benchmarks, Lisa2 Mona outperforms Lydia and Lisa2-Spot is comparable to/better than its fair counterparts.

Lisa2 comprehensively outperforms its fair counterparts on structured benchmarks. Lisa2-Spot solves almost twice as many benchmarks that its fair counterparts while Lisa2-Mona solves ∼40% more benchmarks than Lydia. This highlights the benefits of our optimizations on realistic benchmarks. More broadly, it reflects the merits of identifying and leveraging patterns appearing in structured (realistic) formulas.

Next, we examine each structured benchmark class in detail.

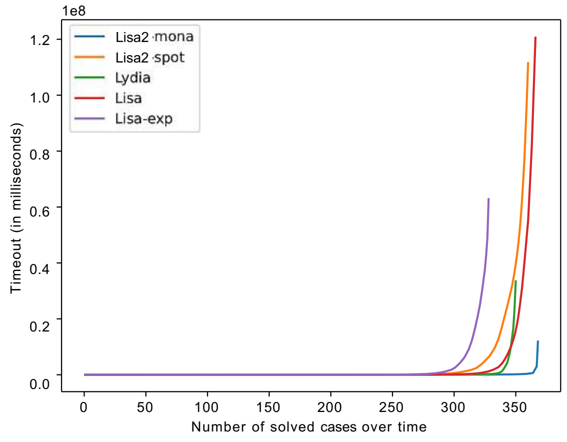
Lisa2 performs remarkably on the nim benchmarks.: Both variants of Lisa2 outperform their fair counterparts by a large margin in both, the number of benchmarks solved and runtime. Lisa2 solves almost twice as many benchmarks as their counterparts (Figure 5). Furthermore, the runtime speedup ranges between 2x-8000x with most benchmarks displaying greater than 500x speedup on Mona's ShMTBDD data structure; and on average 5x speedup on Spot's labeled-graph and ROBDD data structure (Table I).

This outcome is impressive as the nim-benchmarks had proven to be challenging for prior compositional approaches. This occurs since on these benchmarks the intermediate (minimal) DFAs tend to be very large even though final (mininal) DFA is quite small. Hence, it is not uncommon for Lisa or Lydia to fail at an intermediate stage due to memory or time shortage. We attribute Lisa2's success to our optimizations in reducing the number of compositional steps. For most benchmarks in the nim-class, after duplication removal and semantic transformations, the resulting DAG comprised of 5-20% fewer compositions steps than the AST obtaining from unrolling on boolean operators only. In another class of nim-benchmarks derived from [20], this reduction even ranged between 20-50%, resulting in better performance gain.

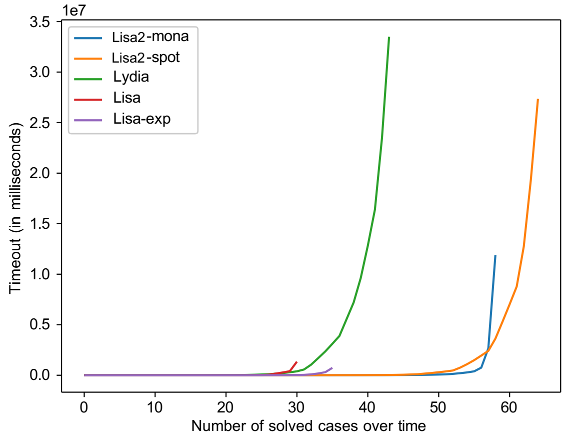
These experiments clearly demonstrate the advantage of reducing the composition steps.

Performance on counter benchmarks: On the single- and double-counter classes of benchmarks, Lisa2-Mona demonstrates 1.4x-100x runtime improvement over Lydia. Whereas, Lisa2-Spot displays 1.5x-100x runtime improvement over Lisa but is sometimes slower than Lisa-Explicit, as demonstrated in Table II.

We attribute the performance of Lisa2 on the counter



(a) Cactus plot: All benchmarks. Timeout 4hrs



(b) Cactus plot: Structured benchmarks. Timeout 4hrs

Fig. 4: Overall Performance

[3]SYNTCOMP: https://www.syntcomp.org. Benchmarks were taken from https://github.com/whitemech/finite-synthesis-datasets/tree/main. We chose benchmarks from the whitemech repository because (a). All SYNTCOMP23 benchmarks in LTLf track were obtained by converting the whitemech benchmarks to TLSF format, (b). All baseline tools natively support the format used in whitemech as opposed to the TLSF format used by SYNTCOMP.

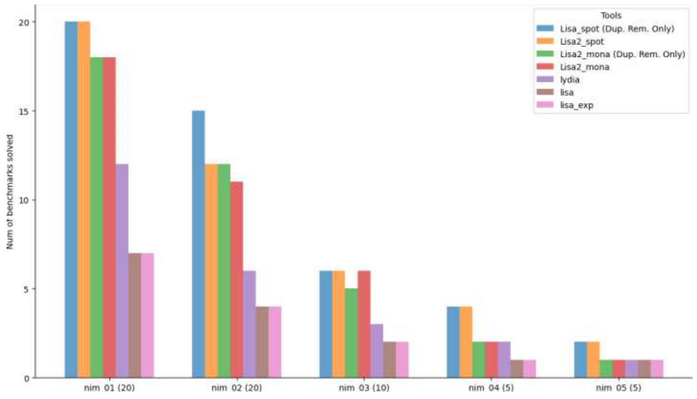


Fig. 5: Number of benchmarks solved on the nim class. x- and y-axes denote benchmark class (total instances in brackets) and num. of benchmarks solved, respectively.

Cases	Lisa2-Mona	Lydia	Lisa2-Spot	Lisa	Lisa-exp
nim_01_1	10	21	10	0	0
nim_01_2	30	44	20	0	0
nim_01_3	30	89	40	10	10
nim_01_4	50	277	50	20	10
nim_01_5	50	1087	80	50	40
nim_01_6	80	4413	100	120	100
nim_01_7	90	16929	110	210	220
nim_01_8	120	64115	160	-	-
nim_01_9	190	181994	200	-	-
nim_01_10	230	780581	240	-	-
nim_01_11	380	1652381	330	-	-
nim_01_12	420	3203658	360	-	-
nim_01_13	620	-	560	-	-
nim_01_14	820	-	780	-	-
nim_01_15	2220	-	910	-	-
nim_01_16	9780	-	810	-	-
nim_01_17	49100	-	810	-	-
nim_01_18	-	-	1300	-	-
nim_01_19	108890	-	1490	-	-
nim_01_20	-	-	1820	-	-
nim_02_1	20	69	30	20	10
nim_02_2	40	483	80	60	80
nim_02_3	80	5979	180	380	400
nim_02_4	130	92548	380	2040	1760
nim_02_5	200	650682	820	-	-
nim_02_6	350	3574672	2430	-	-
nim_02_7	480	-	8890	-	-
nim_02_8	1150	-	76890	-	-
nim_02_9	2540	-	348840	-	-
nim_02_10	3950	-	466940	-	-
nim_02_11	-	-	-	-	-
nim_02_12	17690	-	452700	-	-
nim_02_13	-	-	-	-	-
nim_02_14	-	-	-	-	-
nim_02_15	-	-	3970670	-	-
nim_03_1	60	465	110	160	160
nim_03_2	160	52220	810	3050	3260
nim_03_3	1110	1653251	27080	-	-
nim_03_4	2110	-	91610	-	-
nim_03_5	4000	-	396780	-	-
nim_03_6	7270	-	1215350	-	-
nim_04_1	180	23813	1050	3920	4520
nim_04_2	2540	7083597	75430	-	-
nim_04_3	-	-	1670340	-	-
nim_04_4	-	-	6634310	-	-
nim_05_1	1310	755274	23500	112100	123220
nim_05_2	-	-	1718010	-	-

TABLE I: Runtime in millisecs for nim. Timeout 4hrs

Cases	Lisa2-Mona	Lydia	Lisa2-Spot	Lisa	Lisa-exp
counter_1	10	7	10	10	10
counter_2	10	17	40	60	60
counter_3	10	33	310	560	540
counter_4	10	57	20	10	10
counter_5	20	82	20	30	10
counter_6	50	156	50	60	30
counter_7	200	371	170	300	100
counter_8	810	1111	690	9560	290
counter_9	3520	4212	2870	125350	990
counter_10	15190	16611	14770	113410	3700
counter_11	66390	76345	61690	-	15130
counter_12	366250	474631	277750	-	90000
counter_13	1910580	2419211	1762580	-	430210
counter_14	9241030	10013917	7970190	-	-
counter_15	-	-	-	-	550
counters_1	10	17	50	80	70
counters_2	10	55	10	10	10
counters_3	20	150	30	80	30
counters_4	80	1103	130	1220	170
counters_5	600	25784	1000	29540	1060
counters_6	7850	660391	8090	941940	7380
counters_7	74590	-	60060	-	47080

TABLE II: Runtime in millisecs for counters. Timeout 4hrs.

benchmarks to the unrolling depth. This is because for most of these benchmarks, duplication removal and semantic transformation did not result in any significant reduction in composition steps as the benchmarks exhibit neither multiple occurrences of a subformula nor are their patterns amenable to the syntactic transformation. We observed that Lydia would fail because of the accumulation in number and size of intermediate DFAs in its AST that unrolls till the literals. This is aggravated by the ShMTDD datastructure to represent DFAs as they can become memory extensive. On the other hand, on these benchmarks, Lisa and Lisa-Explicit benefit from the shallowest unrolling. Lisa2-Spot unrolls the formula deeper than Lisa and Lisa-Explicit, resulting in many more composition steps. The runtime of Lisa2-Spot compared to Lisa-Explicit is further affected as the underlying datastructure of Spot's labeled graphs and ROBDDs are known to result in slower compositions.

A closer examination of this class of benchmark revealed a potential avenue for improvement. While the formulas did not have duplicates, they had several *symmetric subformulas* upto propositional isomorphism. Further optimizations based on leveraging such similarities within such formulas could further improve the performance of LTLf-to-DFA conversion tools, including ours.

Lisa2-Spot vs. Lisa2-Mona.: We compare the performance of Lisa2-Spot and Lisa2-Mona. Note that here the underlying algorithm is identical. The only difference between the two is the choice of datastrcture for DFA representations. As a result, we expect this experiment to highlight the impact of datastructure on a tool's performance.

Our observations confirm that the datastructure plays a vital role in a tool's performance, as we observe that the tools Lisa2 Mona and Lisa2-Spot display the same differences displayed by the underlying datastrcuture, i.e. the observed trend is that Lisa2-Mona consumes more memory but is faster while Lisa2-Spotmay be slower but it consumes lesser

Cases	Lisa2-Spot	Lisa2-Spot (Dup. Rem. only)	Lisa2-Mona	Lisa2-Mona (Dup. Rem. only)
nim_01_01	**10**	20	10	10
nim_01_02	20	20	30	**20**
nim_01_03	40	40	**30**	40
nim_01_04	**50**	60	50	50
nim_01_05	80	**70**	**50**	70
nim_01_06	100	100	**80**	90
nim_01_07	**110**	120	**90**	110
nim_01_08	160	**150**	**120**	140
nim_01_09	200	**170**	**190**	230
nim_01_10	240	**210**	230	**210**
nim_01_11	330	**290**	380	**280**
nim_01_12	360	**350**	**420**	440
nim_01_13	560	**500**	**620**	640
nim_01_14	780	**570**	**820**	870
nim_01_15	910	**690**	2220	**1750**
nim_01_16	810	810	9780	**4270**
nim_01_17	810	**780**	**49100**	50760
nim_01_18	1300	**1170**	-	-
nim_01_19	1490	**1400**	108890	**81370**
nim_01_20	1820	**1700**	-	-
nim_02_01	**30**	40	**20**	30
nim_02_02	80	80	40	40
nim_02_03	180	180	**80**	90
nim_02_04	**380**	400	**130**	140
nim_02_05	820	**770**	**200**	220
nim_02_06	2430	**2240**	350	**330**
nim_02_07	8890	**4870**	**480**	510
nim_02_08	76890	**66990**	1150	1150
nim_02_09	348840	**276800**	2540	**2380**
nim_02_10	**466940**	4576480	**3950**	5000
nim_02_11	-	2953960	-	6930
nim_02_12	452700	**299940**	**17690**	18400
nim_02_13	-	13147500	-	-
nim_02_14	-	-	-	-
nim_02_15	**3970670**	4717260	-	-
nim_03_01	110	**100**	60	60
nim_03_02	810	**700**	**160**	170
nim_03_03	27080	**8580**	1110	**640**
nim_03_04	**91610**	173470	2110	**1800**
nim_03_05	396780	**314470**	**4000**	5810
nim_03_06	**1215350**	9473690	7270	-
nim_04_01	1050	**890**	**180**	210
nim_04_02	**75430**	104110	2540	**2120**
nim_04_03	1670340	**1093050**	-	-
nim_04_04	**6634310**	14091500	-	-
nim_05_01	23500	**17230**	1310	**1070**
nim_05_02	**1718010**	5659390	-	-

TABLE III: Ablation Study: Runtime in millisecs for nim benchmarks on Lisa2 and its version with the duplicate removal optimization (i.e. no semantic transformation) only. The **lower runtime** is in bold. Timeout 4hrs.

memory, hence is capable to solve more benchmarks.

These observations further highlight the need for fair comparisons in LTLf-to-DFA conversions that we raised earlier, hence reflects on the importance of tools supporting both datastrucutres for DFA representation.

C. Ablation Study

Finally, we perform an ablation study to examine the effect of each optimization individually. Together the optimizations of duplicate removal and syntactic transformation reduce the number of composition operations. We are interested in studying their effects individually. For this, we compare the performance of Lisa2 (under each DFA datastructure choice) against its own version in which the syntactic transformation has been disabled, i.e., they only applied duplicate removal.

Figure 5 demonstrates the performance of Lisa2-Spot and Lisa2-Mona against their variants that perform duplicate removal only. Apriori, one would imagine that compounding reduction in composition steps through duplicate removal and syntactic transformation would result in improved performance (both in number of benchmarks solved and runtime). However, Figure 5 demonstrates that in some cases (nim_02) the variant that only performed duplicate removal solved more benchmarks. This surprising result led us to further examine the runtime of these tools, shown in Table III. This reveals that there are a significant number of cases where only performing duplicate removal performed better than compounding both optimizations and there are equally many cases where compounding optimizations displayed the stronger runtime performance. In either case, the overall runtime performance is still an improvement over prior state-of-the-art tools.

In order to understand this behavior, we first ascertained that each optimization consumes such a negligible amount of time that it cannot be considered to be the reason behind runtime decline. Similarly, we ascertained that each optimization contributed to reducing the number of composition steps.

We conclude that the unpredictability, despite the reduction in number of composition steps, arises due to the creation of new nodes (new subformulas) after syntactic transformation. To elaborate further, syntactic transformations may result in creating subformulas that were not originally present in the input formula. It is possible that the new formulas are such that even though their DFA construction may require fewer composition steps, these steps may be more expensive as an intermediate DFA may be difficult to create. This results in unpredictability in performance when both optimizations are compounded. In contrast, duplicate removal never creates any new node (new subformula). It only reduces the number of times some nodes may be computed. Hence, duplicate removal will always reduce the overall runtime.

VIII. Concluding Remarks

This work presents Lisa2 which incorporates a series of simple-yet-effective optimizations for compositional approaches for LTLf-to-DFA conversion. Empirical evaluations of this tool displays significant performance improvement, especially on structured benchmarks derived from real-world scenarios: Lisa2 solves $\sim$50% more benchmarks and shows runtime improvement in the range of 1.5x-8000x.

Our optimizations are based on reducing the number of composition steps required to construct the desired DFA. Despite the remarkable performance of Lisa2, our experiments reveal that simply reducing the number of composition steps may not be sufficient, especially if the reduction is accompanied with the creation of new subformulas for which DFA construction may be hard to generate.

We also emphasize on the need for fair comparison to compare algorithmic advances. This is crucial for LTLf-to-

DFA conversion as the choice of datastructure for DFAs have a significant impact on a tool's performance.

Acknowledgements: We thank Marco Favorito and Kuldeep Meel for their help in setting up baseline tools. We thank the anonymous reviewers for their valuable feedback. This work has been supported by the EPSRC through grant EP/X021513/1 and Georgia Institute of Technology's Presidential Undergraduate Research Award for Fall 2023.

References

[1] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *IJCAI*. AAAI Press, 2013, pp. 854–860.

[2] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE, 1977, pp. 46–57.

[3] A. Camacho, E. Triantafillou, C. Muise, J. Baier, and S. McIlraith, "Non-deterministic planning with temporally extended goals: Ltl over finite and infinite traces," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, no. 1, 2017.

[4] G. De Giacomo, F. M. Maggi, A. Marrella, and F. Patrizi, "On the disruptive effectiveness of automated planning for ltlf-based trace alignment," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[5] J. A. Baier and S. McIlraith, "Planning with temporally extended goals using heuristic search," in *ICAPS*. AAAI Press, 2006, pp. 342–345.

[6] M. Lahijanian, S. Almagor, D. Fried, L. E. Kavraki, and M. Y. Vardi, "This time the robot settles for a cost: A quantitative approach to temporal logic planning with partial satisfaction." in *AAAI*. AAAI Press, 2015, pp. 3664–3671.

[7] R. Brafman, G. De Giacomo, and F. Patrizi, "LTLf/LDLf non-markovian rewards," in *AAAI*, vol. 32, no. 1, 2018.

[8] A. Camacho, R. T. Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith, "LTL and beyond: Formal languages for reward function specification in reinforcement learning." in *IJCAI*, vol. 19, 2019, pp. 6065–6073.

[9] K. Jothimurugan, S. Bansal, O. Bastani, and R. Alur, "Compositional reinforcement learning from logical specifications," *Advances in Neural Information Processing Systems*, vol. 34, pp. 10 026–10 039, 2021.

[10] M. Pesic, D. Bosnacki, and W. M. P. van der Aalst, "Enacting declarative languages using LTL: avoiding errors and improving performance," in *SPIN*. Springer, 2010, pp. 146–161.

[11] S. Bansal, Y. Li, L. M. Tabajara, M. Y. Vardi, and A. Wells, "Model checking strategies from synthesis over finite traces," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2023, pp. 227–247.

[12] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," in *CAV*. Springer, 1999, pp. 172–183.

[13] S. Bansal, Y. Li, L. Tabajara, and M. Vardi, "Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications," in *AAAI*, vol. 34, no. 06, 2020, pp. 9766–9774.

[14] G. De Giacomo and M. Favorito, "Compositional approach to translate LTLf/LDLf into deterministic finite automata," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 31, 2021, pp. 122–130.

[15] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko, "From spot 2.0 to spot 2.10: What's new?" in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13372. Springer, 2022, pp. 174–187.

[16] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm, "Mona: Monadic second-order logic in practice," in *TACAS*. Springer, 1995, pp. 89–110.

[17] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[18] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *PSTV*. Springer, 1995, pp. 3–18.

[19] N. Klarlund and A. Møller, *MONA Version 1.4: User Manual*, Jan 2001.

[20] L. M. Tabajara and M. Y. Vardi, "Partitioning techniques in LTLf synthesis," in *IJCAI*. AAAI Press, 2019, pp. 5599–5606.

Formal Methods in Computer-Aided Design 2024

Clausal Equivalence Sweeping

Armin Biere
University Freiburg
Freiburg, Germany
biere@cs.uni-freiburg.de

Katalin Fazekas
TU Wien
Vienna, Austria
katalin.fazekas@tuwien.ac.at

Mathias Fleury
University Freiburg
Freiburg, Germany
fleury@cs.uni-freiburg.de

Nils Froleyks
Johannes Kepler University
Linz, Austria
nils.froleyks@jku.at

***Abstract*—The state-of-the-art to combinational equivalence checking is based on SAT sweeping. It recursively establishes equivalence of internal nodes of two circuits to prove equivalence of their outputs. The approach follows the topological order from inputs to outputs and makes use of simulation to refine the set of potentially equivalent nodes to reduce the number of SAT solver queries. This non-uniform hybrid reasoning, using both the circuit structure and a clausal encoding, is complex to orchestrate. In earlier work, clausal encoding was avoided using a dedicated circuit-aware SAT solver. Instead, we propose to perform SAT sweeping directly on the clausal encoding of the complete equivalence checking problem within the SAT solver, but again relying on a second, dedicated, internal SAT solver. Both SAT solvers work on a clausal representation. This allows to transparently make use of all the advanced reasoning capabilities of the SAT solver, particularly pre- and inprocessing techniques.**

***Index Terms*—Equivalence Checking, SAT Sweeping, Miters, Equivalence Reasoning, Conjunctive Normal Form, Backbones.**

I. Introduction

Hardware equivalence checking is considered one of the oldest and most successful industrial formal verification techniques. Its purpose is to formally prove that a synthesized circuit matches its golden model. While early approaches [1] relied on binary decision diagrams (BDDs), more recent approaches rely on SAT sweeping [2]. It is fair to assume that SAT sweeping is important in commercial equivalence checking tools too.

The state-of-the-art [3]–[8] uses a *hybrid approach* to detect equivalent literals through SAT sweeping. It follows the topological structure of the two compared circuits and uses incremental queries to the SAT solver as well as dedicated SAT solving engines which can be made aware of the circuit structure too [4]–[6]. It can also focus the SAT solving effort on small parts of the circuit, which avoids the overhead in having the SAT solver deal with the full combined problem.

This hybrid approach is in contrast to a *monolithic approach*, advertised in this paper, in which the equivalence checking problem (the miter [9]) is translated once as a whole into a clausal representation in conjunctive normal form (CNF) and then simply given to a SAT solver. The advantage of the monolithic approach is that it can easily make use of sophisticated preprocessing [10] and inprocessing [11] techniques implemented in modern SAT solvers.

This work was supported by the state of Baden-Württemberg through bwHPC, the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG, by the Austrian Science Fund (FWF) under project No. T-1306, and by a gift from Intel Corporation.

In this paper we combine the benefits of both approaches by using within the main SAT solver (KISSAT) a second embedded simple SAT solver (KITTEN) for SAT sweeping directly on CNF. This not only improves monolithic solving of miters substantially but also reduces solving time on other formulas in CNF for which no circuit structure is available.

Hybrid SAT sweeping relies on *structural hashing* to remove isomorphic parts of the miter. For instance, when comparing two identical copies of the same circuit, structural hashing alone can prove equivalence. In the monolithic approach this is much harder, at least for plain CDCL solving [10], which empirically fails on such isomorphic miters [12], [13].

Our recent work [14] on *clausal congruence closure* allows to simulate structural hashing on the CNF level. It relies on gate extraction and succeeds in solving such simple isomorphic miters instantly. Alone it falls far behind hybrid approaches on more practically relevant miters checking equivalence of optimized (synthesized) and original (golden) circuits, unless it is combined with *clausal equivalence sweeping*, presented in this paper. This monolithic sweeping approach has not been described nor evaluated in the literature before, except briefly being mentioned in system descriptions of KISSAT in SAT competition proceedings [15], [16]. For more related work from the SAT and CP community see [14], [17], [18].

II. Preliminaries

We assume the reader is familiar with standard notations in SAT, i.e., we work with formulas F in conjunctive normal form (CNF), usually denoted as a set $F = \{C_1, \ldots, C_m\}$. Each clause C is a set of literals $C = \{l_1, \ldots, l_n\}$, with each literal l being a variable v or its negation $\bar{v}$. We also use logical notation $F = C_1 \wedge \cdots \wedge C_m$ and $C = l_1 \vee \cdots \vee l_n$ as well as logical negation $\neg v = \bar{v}$ and assume $\neg\neg l = l$. Besides variables we also use the Boolean constants $\mathbb{B} = \{0, 1\}$.

The variables are taken from a fixed set $\mathcal{V}$, which we assume to be totally ordered via $\leq$. The variable v of a literal l is obtained as $|l| = v$, meaning $l = v$ or $l = \neg v$. The variable order yields a preorder on the set of all literals, denoted as $\mathcal{L}$, and the Boolean constants as follows: $l \leq l'$ iff $|l| \leq |l'|$ and $0, 1 \leq l$ for all $l, l' \in \mathcal{L}$ (note $l \leq \neg l$ and $\neg l \leq l$). We also use the irreflexive version $<$, where additionally $|l| \neq |l'|$.

With $\mathcal{V}(C) = \{|l| \mid l \in C\}$ and $\mathcal{V}(F) = \{\mathcal{V}(C) \mid C \in F\}$ we denote the set of variables of a clause and a formula and similarly for $\mathcal{L}(C)$ and $\mathcal{L}(F)$ for its literals. Let $|S|$ refer to the number of elements of a set S.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_29

```
full-sweeping (CNF $G$)
literal representative $\rho\colon \mathcal{L} \to \mathcal{L} \cup \mathbb{B}$ with $\rho(l) = l$
if $G$ is unsatisfiable return $\lambda l.(l \neq |l|)$    // map $v \mapsto 0$
pick initial assignment $\sigma$ with $\sigma(G) = 1$
backbone candidates $B \leftarrow \{l \in \mathcal{L}(G) \mid \sigma(l) = 1\}$
equivalent literals partition $P \leftarrow \{B\}$
while $B \neq \emptyset$
    pick $l \in B$ and set $B \leftarrow B \backslash \{l\}$
    if exists model $\sigma$ with $\sigma(G \wedge \neg l) = 1$    // SAT call
        $B \leftarrow \{l \in B \mid \sigma(l) = 1\}$    // refine backbone
        $P \leftarrow$ refine ($P$, $\sigma$)    // refine partition
    else
        $\rho \leftarrow$ propagate ($G$, $\rho \circ \{l \mapsto 1\} \circ \{\neg l \mapsto 0\}$)
        remove from $L \in P$ all $l \in L$ with $\rho(l) \in \mathbb{B}$
        $G \leftarrow \rho(G)$
while exists literal class $L \in P$ with $|L| > 1$
    pick $k, l \in L$ with $k < l$
    if exists $\sigma$ with $\sigma(G \wedge l \wedge \neg k) = 1$ or    // SAT call
                    $\sigma(G \wedge \neg l \wedge k) = 1$    // SAT call
        $P \leftarrow$ refine ($P$, $\sigma$)    // refine partition
    else
        $\rho \leftarrow \rho \circ \{l \mapsto k\} \circ \{\neg l \mapsto \neg k\}$
        remove $l$ from $L$ in $P$
return $\rho$
```

Fig. 1: Pseudo code of our full SAT sweeping routine, which in practice is only applied to variable environments (*cf.* Fig.4/5). We use the '$\circ$' operator to denote function composition.

An assignment $\sigma\colon \mathcal{V} \to \mathbb{B}$ is extended to literals, clauses and formulas by applying Boolean simplification. A formula F is *satisfiable* if there is an assignment σ with $\sigma(F) = 1$, also called *satisfying assignment* or *model*. Let $\bot = \{\emptyset\}$ denote the unsatisfiable CNF consisting of the empty clause $\emptyset$. Given a satisfiable formula F, a literal l is a *backbone* of F if $\sigma(l) = 1$ for all models σ of F. This can be checked by showing that $F \wedge \neg l$ is unsatisfiable. Two literals k and l are *equivalent* if $\sigma(k) = \sigma(l)$ in all models σ of F. This can be checked by showing that $F \wedge l \wedge \neg k$ is unsatisfiable as well as $F \wedge \neg l \wedge k$.

III. Algorithm

Our unbounded algorithm for *full-sweeping* is shown in Fig. 1. It returns a *mapping* ρ of the literals L of the given formula to literals or to Boolean constants $\mathbb{B} = \{0, 1\}$. We further assume $\rho(\neg l) = \neg\rho(l)$ and $\rho(l) \leq l$ as it is common in this kind of union-find data-structure.

The entire sweeping algorithm has three phases. First, lines 1–5, it tries to find a satisfying assignment σ. If none exists, an arbitrary constant mapping ρ is returned, i.e., $\rho(v) = 0$ for all $v \in \mathcal{V}$, guaranteed to falsify at least one clause. Otherwise, σ is used to determine the backbone candidates B (literals set to true) and an initial equivalence class of literals also all set to true. Note that thereby negations of these literals are also considered potentially equivalent with each other.

```
refine ($P$, $\sigma$)
$R \leftarrow \emptyset$
for all $L \in P$
    $L_i \leftarrow \{l \in L \mid \sigma(l) = i\}$ for $i \in \mathbb{B}$
    if $L_0 = \emptyset$ or $L_1 = \emptyset$ then $R \leftarrow R \cup \{L\}$
    else $R \leftarrow R \cup \{L_0\} \cup \{L_1\}$
return $R$
```

Fig. 2: Refinement of equivalent literal partition.

```
propagate (CNF $F$, literal mapping $\rho\colon \mathcal{L} \to \mathcal{L} \cup \mathbb{B}$)
$F \leftarrow \rho(F)$    // pre-simplify
while $\emptyset \notin F$ and there is a unit clause $\{l\} \in F$
    $\rho \leftarrow \rho \circ \{l \mapsto 1\} \circ \{\neg l \mapsto 0\}$
    $F \leftarrow \rho(F)$    // simplify
return $\rho$
```

Fig. 3: Unit propagation on a literal mapping.

In the second phase, lines 6–14, each remaining l in the backbone candidate set B is checked to have a model of the formula falsifying l. If such a model exists, we remove all falsified literals from B in that model and refine the equivalence classes in the partitioning according to Fig. 2 by splitting classes inconsistent with that model into one class of literals assigned to 0 and one class of literals assigned to 1. Otherwise, there is no model of the formula which sets the considered backbone candidate l to 0. So we update ρ by setting l permanently to 1 and $\neg l$ to 0 and *propagate* this information over the formula, as shown in Fig. 3, which might deduce additional constant assignments. Afterwards, the formula is simplified (line 14) by applying the updated mapping.

In the third phase, lines 15–22, after backbone extraction, we check for each pair of remaining equivalent literals candidates, within the same equivalence class, whether there exists a model of the formula with the two literals assigned to different values. If this is the case we split their equivalence class as well as all other equivalence classes which are inconsistent with the model. Otherwise, we have shown that these two literals are equivalent and record that information by mapping the larger literal to the smaller (and accordingly their negations).

This procedure calls a SAT oracle in three places and as such is not really useful to simplify a formula for which we only want to know whether it is satisfiable. Thus in order to use this sweeping procedure in the context of SAT solving, we have to limit the effort put into these SAT calls. There are two obvious ways to achieve this. Either we replace the oracle calls by some cheaper procedure to limit the run-time of the oracle or we apply full sweeping only to a subset of literals.

We have explored the first option before in LINGELING [19] and SPLATZ [20] without much success though. Therefore, KISSAT uses the second approach, shown in Fig. 4. As in hybrid approaches [4]–[6], we focus each full-sweeping only on a small part of the formula, assuming that the cheap-to-detect equivalences are between literals close to each other.

```
bounded-sweeping (CNF $F$, bound $k \in \mathbb{N}$)
1  working set $\Gamma \leftarrow \mathcal{V}(F)$    // all variables in $F$
2  while $\Gamma \neq \emptyset$
3      pick $v \in \Gamma$ and set $\Gamma \leftarrow \Gamma \backslash \{v\}$
4      $G \leftarrow$ environment ($\{v\}$, $\emptyset$, $F$, $k$)
5      $\rho \leftarrow$ full-sweeping ($G$)    // sweep environment clauses
6      $\rho \leftarrow$ propagate ($F$, $\rho$)    // propagate $\rho$ on whole $F$
7      $F' \leftarrow \rho(F)$                 // simplify $F$ with $\rho$
8      if $\emptyset \in F'$ return $\bot$  // return CNF with empty clause
9      $\Gamma \leftarrow \Gamma \cup \mathcal{V}(F' \backslash F)$  // add variables in changed clauses
10     $F \leftarrow F'$
11 return $F$
```

Fig. 4: Pseudo code of our bounded SAT sweeping routine.

```
environment (variables $V$, CNF $G$, CNF $F$, bound $k \in \mathbb{N}$)
1  if $k = 0$ return $G$
2  $G' \leftarrow \{C \in F \mid V \cap \mathcal{V}(C) \neq \emptyset\}$    // clauses with $V$
3  $V' \leftarrow \bigcup \mathcal{V}(G')$           // variables in clauses with $V$
4  return environment ($V'$, $G'$, $F$, $k-1$)
```

Fig. 5: Compute bounded environment of a variable.

To this extent we consider two variables (and thus their literals) to be "very close" if they occur in the same clause and extend this notion recursively in a breadth-first manner limited by some bound k, i.e., the distance between two variables. We further decided to restrict the part of a formula to which full sweeping is applied to consist of all clauses containing variables a maximum distance away from a given variable v, i.e., the *environment* of v as shown in Fig. 5.

The *bounded-sweeping* algorithm goes over all variables v of the formula and performs full sweeping on the environment of v. The whole formula is then simplified by applying the mapping ρ obtained from the full sweeping of the environment (line 7). All variables in clauses that changed during that simplification are reconsidered (line 9). This approach is sound as both local backbones and equivalences of a sub-formula are of course also backbones and equivalences of the whole formula. It is obviously not complete but gives substantial improvements in practice, as our experiments will show.

IV. Implementation

The use of a dedicated light-weight SAT solver in hybrid approaches (*cf.* [6]) provided the motivation to explore using a separate light-weight *little* SAT solver (KITTEN) within our full-blown state-of-the-art *big* SAT solver (KISSAT). This allows to (i) solve only parts of a big formula by copying it, and (ii) avoids any other interaction of solving the small problems such as keeping statistics, variable scores etc. untouched, and (iii) allows to record proofs in memory in case it becomes necessary to export proofs from the small to the big solver, without the need to support this feature in the big solver.

Although clausal equivalence sweeping, presented in this paper, was the first application of KITTEN inside KISSAT, it was also used to mine definitions [21], [22], and to improve bounded variable elimination [23]. The article on definition-mining [21] contains implementation details about KITTEN.

Most of the time SAT calls during sweeping in Fig. 1 are of course satisfiable as otherwise the formula would radically simplify. Even though often trivial to solve (few or no conflicts), these satisfiable queries are relatively expensive, as a full model of the environment has to be constructed, i.e., at least linear in the number of variables in the environment. Motivated by the usefulness of model rotation in MUS extraction [24], we added an API call "`kitten_flip_literal`" to KITTEN, which checks after a model has been found, whether the value of a single literal can be flipped, without falsifying any clause.

Flipping can be implemented efficiently by traversing only the clauses watched by that literal, an insight which helped to improve stand-alone backbone extraction [25] (after porting it to CADICAL). Without using watches, model rotation appears to be too costly [26]. In our implementation of clausal equivalence sweeping, we aggressively use literal flipping whenever we find a model (at line 9 and 19 in Fig. 1) to refine both backbone candidates and the equivalent literal classes, i.e., any backbone candidate and any literal in an equivalence literal class can be removed if it can be flipped. This technique reduces the number of full satisfiable queries substantially.

Despite a small bound of distance $k = 3$ (which actually starts at $k = 2$ and only is increased to $k = 3$ in the next inprocessing round after successful completion of sweeping), we also limit the number of variables ($2^{13} = 8192$) and clauses ($2^{15} = 32768$) allowed in an environment. Still, also in contrast to clausal congruence closure [14], clausal equivalence sweeping is too costly to run until completion on larger instances. Therefore we limit the effort (time spent in KITTEN measured in "ticks" – an approximation of cache line accesses) relative to the time spent during CDCL, as with other inprocessing procedures, preempt sweeping and continue later with the remaining variables in the next inprocessing round.

V. Benchmarks

We evaluated our approach on problems of the hardware model checking competition (HWMCC) from 2012 [27] and 2020 [28] and from the SAT competition 2022 and 2023. The AIGER [29] problems from HWMCC are encoded into CNF based on detecting and encoding XOR and ITE gates in an optimized way instead of the default AND gate encoding.

Moreover, each miter comes in two versions: iso and opt. The former (iso) compares each circuit with an *isomorphic* copy of itself, while the latter set (opt) uses the command `dc2` of ABC to *optimize* the original circuit and then compares this optimized circuit with the original circuit [30]–[32].

Evaluating our technique on SAT competition benchmarks allows us to assess the potential overhead and benefits of our approach on more general SAT instances that may have less underlying structure which can be exploited by our technique.

VI. Experiments

We implemented our approach in KISSAT and evaluated its performance on the bwForCluster Helix, utilizing AMD Milan

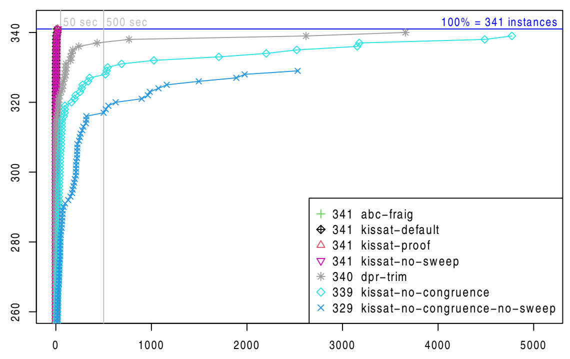


Fig. 6: Number of solved **isomorphic** miters (y-axis) from 341 HWMCC'12 benchmarks in the given time (x-axis in seconds). The legend displays the number of solved instances per solver.

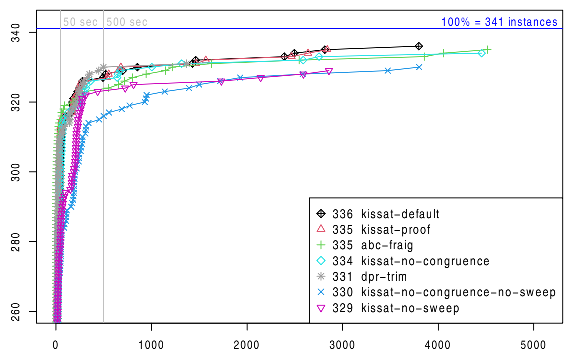


Fig. 7: Number of solved **optimized** miters (y-axis) from 341 HWMCC'12 benchmarks in the given time (x-axis in seconds). The legend displays the number of solved instances per solver.

EPYC 7513 CPUs, with 15 GB of memory and 5000 second time limit. All plots follow the SAT competition set-up [33] showing the number of solved instances (y-axis) over the time it took to solve them (on the x-axis in seconds). Source code is available at [32] and experimental data at [32], [34].

In our experiments we compare the default configuration of KISSAT (kissat-default), where both SAT sweeping and clausal congruence closure [14] is enabled, to activating only one or none of these techniques. Additionally, we consider runs of the default configuration with proof production enabled (kissat-proof), and present here the required time to check these produced proofs using DPR-TRIM as well (dpr-trim).

Figures 6-9 depict the results of the experiments on the HWMCC problems. Here we compare our approach to the state-of-the-art SAT sweeping technique [6] implemented in ABC, available as `fraig -y` in ABC superseding `fraig -x` used in the `cec` command (according to personal communication with the author of ABC). The results show that our pure SAT-based approach, that sees only the clausal representation of the circuits is able to perform comparable to the hybrid approach specialized in reasoning about circuits. Regarding SAT competition problems, we follow [35] and include SBVA-CADICAL, winner of the SAT Competition 2023. The results in Fig. 10-14 demonstrate that *sweeping* and *congruence closure* both contribute to better performance on these more generic competition problems too. Fig. 13/14 also compare solving times versus checking times with DPR-TRIM.

In Fig. 12 we show results on 5 challenging miters from the IWLS'22 paper [6] (originating from [5]) which introduced the advanced SAT sweeping technique implemented in ABC (through the command "`fraig -y`") as also used in our experiments. Again sweeping gives a substantial improvement in our monolithic approach. Note that one miter "test02" can actually be solved by congruence closure instantly (faster than ABC) and does not need sweeping (*cf.* [14] for details).

We further extracted from the log files [32], [34] the following numbers. The time spent in clausal equivalence sweeping with kissat-default is for hwmcc12/opt on average 13.72 sec (0.00 sec - 636.67 sec) and 21.77% (3.26% - 80.98%), for hwmcc20/opt on average 31.42 sec (0.00 sec - 636.67 sec) and 17.89% (3.26% - 80.98%), for iwls22 on average 64.35 sec (0.00 sec - 104.51 sec) and 9.81% (0.00% - 10.92%), for sc2022 on average 34.75 sec (0.04 sec - 681.70 sec) and 4.52% (0.13% - 29.70%), for sc2023 on average 29.65 sec (0.00 sec - 437.05 sec) and 5.87% (0.06% - 93.72%).

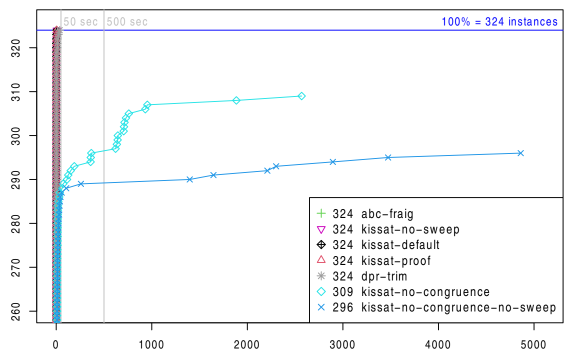


Fig. 8: Number of solved **isomorphic** miters (y-axis) from 324 HWMCC'20 benchmarks in the given time (x-axis in seconds). The legend displays the number of solved instances per solver.

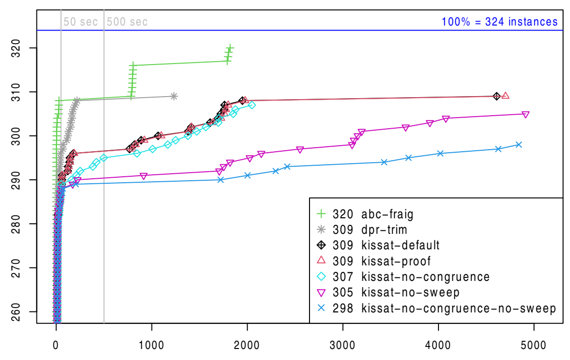


Fig. 9: Number of solved **optimized** miters (y-axis) from 324 HWMCC'20 benchmarks in the given time (x-axis in seconds). The legend displays the number of solved instances per solver.

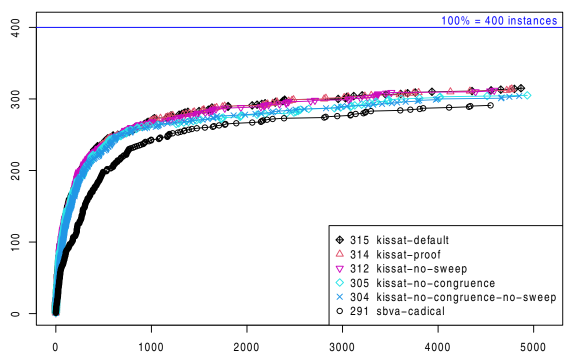


Fig. 10: Number of solved SAT Competition 2022 main track benchmarks (y-axis) in the given time (x-axis in seconds). The legend displays the number of solved instances per solver.

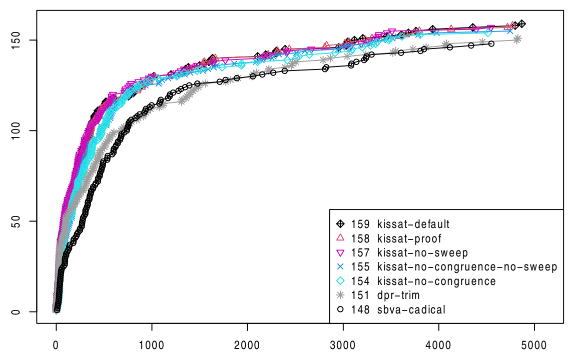


Fig. 13: Number of solved **unsatisfiable** SAT Competition 2022 main track benchmarks (y-axis) in the given time (x-axis in seconds). The total number of unsatisfiable instances is unknown though. The legend displays the number of solved instances per solver.

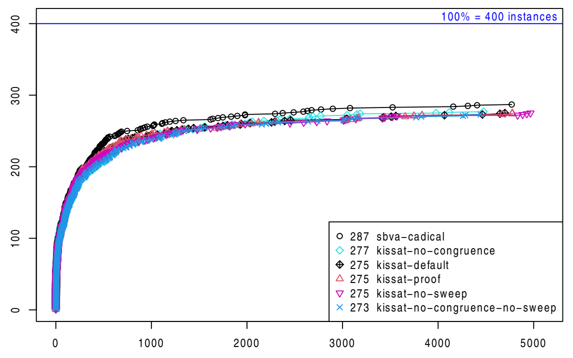


Fig. 11: Number of solved SAT Competition 2023 main track benchmarks (y-axis) in the given time (x-axis in seconds). The legend displays the number of solved instances per solver.

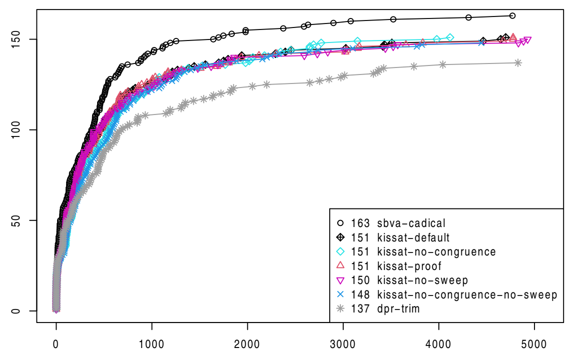


Fig. 14: Number of solved **unsatisfiable** SAT Competition 2023 benchmarks (y-axis) in the given time (x-axis in seconds). The total number of unsatisfiable instances is unknown though. The legend displays the number of solved instances per solver.

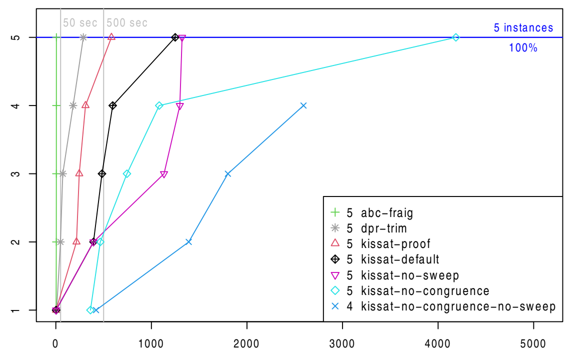


Fig. 12: Number of solved miters (on the y-axis) of the 5 IWLS'22 benchmarks in the given time (on the x-axis in seconds). The legend displays the number of solved instances per solver. One of the miters, i.e., test02, is instantly solved by KISSAT with clausal congruence closure even though it was considered challenging in [6]. This is due normalization during ITE gates extraction. See [14] for a more detailed explanation.

The number of backbones and equivalences found were for hwmcc12/opt 70 780 backbones and 446 784 equivalences, for hwmcc20/opt 12 976 backbones and 162 427 equivalences, for iwls22 2 052 backbones and 58 792 equivalences, for sc2022 298 065 backbones and 1 590 810 equivalences, for sc2023 838 120 backbones and 4 019 861 equivalences.

The percentage of satisfiable queries was for hwmcc12/opt 91.45%, for hwmcc20/opt 95.27%, for iwls22 94.25%, for sc2022 95.64%, for sc2023 96.00% and the ratio of successfully flipped literals over the number of satisfiable queries was for hwmcc12/opt 12.77, for hwmcc20/opt 32.34, for iwls22 21.03, for sc2022 64.05, for sc2023 23.34.

VII. Conclusion

We presented a "big-little" approach to clausal equivalence sweeping directly on CNF using an embedded SAT solver KITTEN inside of KISSAT and show that it improves solving hard equivalence checking problems substantially as well as being useful on plain CNF problems from the SAT competition.

References

[1] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34st Conference on Design Automation, Anaheim, California, USA, Anaheim Convention Center, June 9-13, 1997*, E. J. Yoffa, G. D. Micheli, and J. M. Rabaey, Eds. ACM Press, 1997, pp. 263–268.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, 2002.

[3] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis, "Parallel combinational equivalence checking," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 3081–3092, 2020.

[4] L. G. Amarù, F. S. Marranghello, E. Testa, C. Casares, V. N. Possani, J. Luo, P. Vuillod, A. Mishchenko, and G. D. Micheli, "SAT-sweeping enhanced for logic synthesis," in *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 2020, pp. 1–6.

[5] H. Zhang, J. R. Jiang, L. G. Amarù, A. Mishchenko, and R. K. Brayton, "Deep integration of circuit simulator and SAT solver," in *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 2021, pp. 877–882.

[6] H. Zhang, J. R. Jiang, A. Mishchenko, and L. G. Amarù, "Improved large-scale SAT sweeping," in *Proc. 31st International Workshop on Logic & Synthesis*, 2022.

[7] Z. Chen, X. Zhang, Y. Qian, Q. Xu, and S. Cai, "Integrating exact simulation into sweeping for datapath combinational equivalence checking," in *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*. IEEE, 2023, pp. 1–9.

[8] H. Pan, R. Zhang, Y. Xia, L. Wang, F. Yang, X. Zeng, and Z. Chu, "A semi-tensor product based circuit simulation for sat-sweeping," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2024, Valencia, Spain, March 25-27, 2024*. IEEE, 2024, pp. 1–6.

[9] D. Brand, "Verification of large synthesized designs," in *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, M. R. Lightner and J. A. G. Jess, Eds. IEEE Computer Society / ACM, 1993, pp. 534–537.

[10] A. Biere, M. Järvisalo, and B. Kiesl, "Preprocessing in SAT solving," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 391–435.

[11] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 355–370.

[12] A. Biere, M. Heule, M. Järvisalo, and N. Manthey, "Equivalence checking of HWMCC 2012 circuits," in *Proc. of SAT Competition 2013 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, A. Balint, A. Belov, M. Heule, and M. Järvisalo, Eds., vol. B-2013-1. University of Helsinki, 2013, p. 104.

[13] M. Heule, M. Järvisalo, and A. Biere, "Revisiting hyper binary resolution," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, ser. Lecture Notes in Computer Science, C. P. Gomes and M. Sellmann, Eds., vol. 7874. Springer, 2013, pp. 77–93.

[14] A. Biere, K. Fazekas, M. Fleury, and N. Froleyks, "Clausal Congruence Closure," in *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, ser. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[15] A. Biere, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021," in *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2021-1. University of Helsinki, 2021, pp. 10–13.

[16] A. Biere and M. Fleury, "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022," in *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2022-1. University of Helsinki, 2022, pp. 10–11.

[17] M. Heule and A. Biere, "Blocked clause decomposition," in *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. L. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312. Springer, 2013, pp. 423–438.

[18] M. Codish, Y. Fekete, and A. Metodi, "Backbones for equality," in *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, ser. Lecture Notes in Computer Science, V. Bertacco and A. Legay, Eds., vol. 8244. Springer, 2013, pp. 1–14.

[19] A. Biere, "Lingeling and friends entering the SAT Race 2015," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 15/2, 2015.

[20] ——, "Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016," in *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. University of Helsinki, 2016, pp. 44–45.

[21] M. Fleury and A. Biere, "Mining definitions in Kissat with Kittens," *Formal Methods Syst. Des.*, vol. 60, no. 3, pp. 381–404, 2022.

[22] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, "Moving definition variables in quantified boolean formulas," in *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*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 462–479.

[23] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 61–75.

[24] A. Belov and J. Marques-Silva, "Accelerating MUS extraction with recursive model rotation," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 37–40.

[25] A. Biere, N. Froleyks, and W. Wang, "Cadiback: Extracting backbones with cadical," in *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, ser. LIPIcs, M. Mahajan and F. Slivovsky, Eds., vol. 271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 3:1–3:12.

[26] M. Janota, I. Lynce, and J. Marques-Silva, "Algorithms for computing backbones of propositional formulae," *AI Commun.*, vol. 28, no. 2, pp. 161–177, 2015.

[27] A. Biere, K. Heljanko, M. Seidl, and S. Wieringa, "Hardware model checking competition (hwmcc'12)," 2012. [Online]. Available: https://fmv.jku.at/hwmcc12

[28] A. Biere, N. Froleyks, and M. Preiner, "Hardware model checking competition (hwmcc'20)," 2020. [Online]. Available: https://hwmcc.github.io/2020

[29] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.

[30] A. Biere, K. Fazekas, M. Fleury, and N. Froleyks, "CNF Encoded Isomorphic and Optimized Miters from Hardware Model Checking Competition 2020 Models," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11202461

[31] A. Biere, "CNF Encoded Isomorphic and Optimized Miters from Hardware Model Checking Competition 2012 Models," Mar. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10823128

[32] [Online]. Available: https://cca.informatik.uni-freiburg.de/ces

[33] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "SAT Competition 2020," *Artif. Intell.*, vol. 301, p. 103572, 2021.

[34] A. Biere, K. Fazekas, M. Fleury, and N. Froleyks, "Clausal equivalence sweeping paper logs," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11203283

[35] A. Biere, M. Järvisalo, D. Le Berre, K. S. Meel, and S. Mengel, "The SAT practitioner's manifesto," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4500928

Automatic Verification of Right-greedy Numerical Linear Algebra Algorithms

Carl Kwan
The University of Texas at Austin
Austin, TX, United States of America
carlkwan@cs.utexas.edu

Warren A. Hunt, Jr.
The University of Texas at Austin
Austin, TX, United States of America
hunt@cs.utexas.edu

***Abstract*—We present an automatic verification process for formally proving the correctness of a class of greedy numerical linear algebra algorithms. To demonstrate our methodology, we present theorem prover verifications of LU and Cholesky decompositions. We formalize a framework for reasoning about matrices and matrix algorithms by partitioning matrices and developing a generalized form of the inductive invariant common to this class of greedy algorithms. Our framework enables users to readily verify any algorithm in this class automatically by simply defining the algorithm itself and specifying the class of matrices on which the algorithm performs. Our framework is also adaptable to other greedy numerical linear algebra algorithms. To our knowledge, this is the first automatic approach to verifying an entire class of numerical linear algebra algorithms.**

***Index Terms*—Numerical linear algebra, LU decomposition, Cholesky factorization theorem, Automated theorem proving.**

I. Introduction

Linear algebra is everywhere, permeating across the natural, mathematical, social, applied, and, in particular, computing sciences. The prevalence of linear algebra includes critical applications in which linear algebra computations are used to build modern infrastructure, perform data analysis affecting policy making, engineer control systems, secure information, create scientific models, and develop hardware, software, and cyberphysical systems. Determining the correctness of these linear algebra computations is vital. Numerical linear algebra concerns algorithms designed to perform such computations accurately. However, implementations of such algorithms can still fail with disastrous consequences. The potential human and capital losses due to inadequate numerical implementations necessitates formal methods.

We present a formal method for automatically verifying a class of greedy numerical linear algebra algorithms. We demonstrate the utility of our approach by verifying a particular flavor of the LU and Cholesky decompositions; that is, we verify that the product of the decomposed matrix is the original matrix itself under the appropriate conditions. We choose these specific decompositions because the development or improvement of any new family of numerical linear algebra algorithms typically begins with one of the “three amigos”: LU, Cholesky, or QR decomposition. This makes LU and Cholesky two of the most ubiquitous algorithms in numerical methods. In this paper, we describe our approach to their mechanization.

This work was supported in part by Intel Corporation and Amazon Science.

To the best of our knowledge, any verification of numerical linear algebra algorithms by way of theorem prover is a new area of research. By applying our approach to two of the three amigos, we intend to embark on a significant line of research involving the systematic and portable verification of *families* of numerical linear algebra algorithms. One major novel contribution we make is to identify the level of abstraction appropriate for reasoning with theorem provers. If we implement matrix algorithms and reason at the level of their scalar entries, such as in Algorithm 1, then our proofs would be intractable because the mathematical expressions quickly become too large and unwieldy. Moreover, such an approach is not easily portable to other algorithms, even if they are in the same family. If we reason at too high a level, then verifying instantiated algorithms would require significant user effort, thus reducing automation. In this paper, we apply a partitioned approach to reasoning about matrices and their algorithms, which enable our verification of LU and Cholesky decompositions using the same shared framework.

LU and Cholesky decompositions are vital numerical techniques with broad applications in linear algebra. LU decomposition factorizes a matrix into a product of lower and upper triangular matrices. Cholesky decomposition specifically applies to symmetric positive definite matrices, breaking them down into the product of a lower triangular matrix and its transpose. Let

- LU(A) compute the LU decomposition of A;
- Chol(A) compute Cholesky decompositions of A;
- L_u be the strictly lower triangular part of LU(A), placing 1s on the diagonal;
- U retrieve the upper triangular part of LU(A).
- L be the lower triangular part of Chol(A); and

Specifically, we verify that

1) if every principal leading submatrices of A is nonsingular, then $A = L_uU$; and
2) if A is symmetric positive definite, then $A = LL^T$.

We discuss the conditions on A later. Both LU and Cholesky decompositions facilitate solving linear systems, performing matrix inversions, and calculating determinants efficiently. Cholesky is particularly useful when solving linear least-

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_30

squares problems, performing Monte Carlo simulations, and optimizing quadratic forms. These sorts of decompositions play crucial roles in numerical stability, computational efficiency, and analysing the accuracy of solutions, making them fundamental tools in diverse fields such as physics, engineering, finance, and machine learning.

The particular variants of LU and Cholesky decomposition in this paper are sometimes known as "right-greedy". "Right-greedy" refers to the particular submatrices that are updated in a partitioned representation of the matrix on which an algorithm operates during the main body of a loop or recursion. At each step of the algorithm, we update the nearest submatrices into the desired form. As the algorithm proceeds, the components are updated from left to right[1]. Our method enables the theorem prover verification of a right-greedy algorithm automatically with very few user-provided hints. To discharge the algorithmic proof of correctness using a theorem prover, the only knowledge necessary is the matrix partitioning and a recognizer for the class of matrices on which the algorithm is expected to operate. Our approach is the first to enable the mechanical verification of an entire family of numerical linear algebra algorithms.

We perform our modeling and verification with ACL2, an industrial-strength first-order logic automated theorem prover with support for the real numbers via non-standard analysis [1], [2]. One advantage of using ACL2 is that the structure of the numerical linear algebra algorithms in which we are interested is well suited to ACL2's extensive support for rewriting and induction. Another advantage of using ACL2 is the support for execution via an underlying Lisp interpreter defined within the theorem prover logic. ACL2 is unique among theorem provers in its capability execute code at the speeds of modern programming languages within the theorem prover itself, making our verified numerical linear algebra programs directly applicable to real-world problems. We use the ACL2 language to model commercial linear algebra applications, and we analyze such codes mechanically for their fitness to purpose using the ACL2 theorem prover.

There are very few verification efforts for numerical linear algebra. First, the sheer number of numerical algorithms, even for linear algebra, is daunting, and new ad hoc algorithms for specific applications are often being published. Verifying just the algorithms for critical applications would be an endless affair, requiring large-scale organization and resources. Our work addresses the verification of not just one algorithm or application, but an entire *family* of numerical algorithms. Second, different algorithms can have different structures and correctness criteria. This suggests separate proofs for each individual algorithm. Even identifying structures that are exploitable for verification purposes is challenging. Third, there is an over reliance of indexing in typical presentations of numerical algorithms. Consider the LU decomposition algorithm in Algorithm 1, which is representative of numerical algorithms [3]. Here,

Algorithm 1 Right-greedy LU decomposition (Stewart). [3]

```
for $k = 1 : n-1$ do
  if $A[k,k] = 0$ Error
  $A[k+1:n,k] = A[k+1:n,k]/A[k,k]$
  $A[k+1:n,k+1:n]$
        $= A[k+1:n,k+1:n] - A[k+1:n,k]A[k,k+1:n]$
```

- $A[a,b]$ refers to the scalar in the a-th row and b-th column,
- $A[a:b,c]$ refers to the column vector from row a to row b in column c,
- $A[a,b:c]$ refers to the row vector from column b to column c in row a,
- $A[a:b,c:d]$ refers to the submatrix from row a to row b and from column c to column d.

Indexing obfuscates the design and intent of the algorithm to the point where it is unclear what is a matrix, what is a vector, or even what is a scalar in the main loop. Unbounded proofs for algorithms of this sort can be discharged by induction or rewriting by a general purpose theorem prover. Fourth, ACL2 has extensive support for execution within its logic and numerical linear algebra algorithms are usually designed to be executed. Some theorem provers can generate executable source or machine code but these tend to be unverified and unoptimized, which limits their utility especially since unverified but optimized numerical linear algebra libraries already exists. Fifth, the scope of linear algebra algorithms is sometimes limited to a class of matrices for which the definition is not conducive to formalization. For example, Cholesky decomposition is designed for matrices that are symmetric positive definite, and the usual definition for positive definite involves quantifying over all vectors. We want to avoid quantifiers in the definition of, say, positive definite because they limit the execution of a recognizer for such matrices.

In this paper we provide our solutions to the five challenges described. To address the first two challenges and partially address the third, we take advantage of the Formal Linear Algebra Methods Environment (FLAME) [4]. FLAME is an approach for systematically deriving numerical linear algebra algorithms that circumvents the problem of indexing by representing numerical linear algebra algorithms in terms of operations on the submatrices in a partitioned form of the original matrix. The partitioned form is not only more readable from a human perspective, but also exposes invariants that facilitates ACL2 reasoning and verification. The problem with algorithms in the original FLAME approach is that they are loop-based and mathematical correctness follows from identifying loop-invariants. In our approach, we recast loops into recursions and instead identify generalizable *inductive* invariants, which better aligns with the spirit of ACL2.

To address the last two challenges and finish addressing the third, we develop ACL2 mechanisms to enable automatic reasoning about linear algebra algorithms, define constructive recognizers for the matrices on which these algorithms operate,

[1]Section IV provides a visual treatment of "right-greedy".

and execute them. It is important for these recognizers to be executable because they can also serve as an efficient way to check whether a matrix is part of a solvable problem before performing more costly procedures. Execution is handled natively by ACL2. To reason about matrix algorithms, we develop our own ACL2 rules for partitioning matrices and finding inductive invariants. Another contribution we make is to identify and develop definitions that enable constructive recognizers for matrices that satisfy an algorithm's precondition.

The rest of this paper is organized as follows: first, we discuss the limited existing literature on linear algebra and theorem proving; second, we introduce the basics of ACL2 and linear algebra; third, we motivate our mechanical method for automatically verifying numerical linear algebra algorithms by describing how to verify LU and Cholesky decompositions; fourth, we describe how to generalize the techniques used to verify our two motivating examples; finally, we conclude by summarizing our approach and discussing its immediate application and future work.

II. Related Work

While theories of matrices and vector space exists in ACL2 and other theorem provers, there are practically no theorem prover verifications of numerical linear algebra algorithms. For the ACL2 theorem prover, the closest relevant existing paper of which we are aware is a formalization and proof of correctness for LU decomposition [5]. There has also been ACL2 work on using abstract single threaded objects to compute the column echelon form of a matrix [6]. Other relevant ACL2 papers include formalizations of matrices [7], [8], vectors (both real and abstract) [9], [10], and vector-valued functions [11]. An application of ACL2 matrices is the VWSIM circuit simulator for rapid, single-flux, quantum (RSFQ) circuits, which is written in ACL2 and based on repeatedly solving linear systems of the form $Ax = b$ [12]. However, VWSIM's matrix solver is not ACL2 verified.

Expanding the purview to theorem provers in general, we find formal theories for matrices that either do not support execution or are limited to basic matrix arithmetic operations (e.g., addition, multiplication, etc.). These include Coq, Lean, Isabelle, and HOL4. In the Coq community, there are at least five proposed formal models for basic matrix theory and recent work towards integrating them has been published [13]. There is a Lean proof that positive definite matrices have an LDL decomposition [14]; however, none of the functions involved in the proof are computable. Isabelle formalizations of many matrix procedures, including algorithms for Gauss-Jordan elimination, Schur decomposition, and finding various normal forms, are in the Archive of Formal Proofs but none are natively executable [15]. There is also a HOL4 theory for basic matrix ideas and operations [16]. While many of these theorem provers are excellent at modelling mathematics, they have little to no support for the direct execution of numerical code, making them unsuited to our purposes.

FLAME is a major influence on our work. In addition to describing how to derive families of numerical linear algebra algorithms and demonstrate their correctness based on different loop-invariants, FLAME also provides an alternate partitioning-based framework for backwards error analysis [17]. However, no formal method tools are used in the FLAME project and FLAME algorithms are not recursive. While FLAME derivations of algorithms such as LU and Cholesky decompositions indicate a natural inductive step, its mathematical proofs for the correctness and existence of these decompositions deviate significantly from our approach. Our approach to correctness is to define a recursive variant of the algorithms of interest and then constructively define a recognizer which induces an induction on the partitioning of the matrix. Existence follows because we posit an explicit algorithm which computes the desired decomposition.

No prior theorem-prover-based work supports FLAME-style analyses. ACL2-specified algorithms are our best option. Our work is the first to provide techniques for formally verifying families of executable numerical linear algebra algorithms.

III. ACL2 Basics

Our theorem prover of choice is ACL2, a first-order logic with support for highly automated reasoning by way of extensive rewriting heuristics and induction. ACL2 contains many built-in features and tooling which support software engineering efforts, proof and theory management, user-controlled rewriting, file I/O and parsing of large-scale designs, calling internal and external automated solvers in a sound manner, and much more, all with extensive publicly-available documentation. ACL2 formalizes an applicative subset of pure Common Lisp, which enables ACL2 code to be efficiently compiled and executed.

In ACL2, functions are total, that is, all functions map all objects in the logic. By first-order, we mean quantifiers can only predicate over individuals (though we avoid explicit quantifiers in practice). The return on this restriction is that first-order logic theorem proving is highly developed, semi-decidable, and allows for truly automated reasoning. ACL2 is primarily based on term-rewriting, which is a set of rules for replacing one logical expression with another equivalent expression. Sophisticated heuristics for rewriting and extensive support for automatic induction allows ACL2 to be a highly automated and efficient tool appealing to commercial applications. ACL2 is sometimes referred to as an industrial-strength theorem prover, where it ensures the correctness of critical systems. ACL2 has been deployed to verify industrial-scale hardware designs and software systems at companies such as Intel, AMD, Oracle, Collins Aerospace, IBM, and ARM [18].

Table 1 lists some commonly used ACL2 functions, macros, and commands. A comprehensive description of the built-in ACL2 functions can be found in the ACL2 documentation [19]. Table 2 lists some commonly used ACL2 linear algebra functions which we do not further describe in this paper. We take advantage of some defined primitive matrix functions [7], but define our own functions to support reasoning about numerical linear algebra algorithms, accessing their results, and executing the algorithms themselves. For functions which are central to this paper, such as recognizers for nonsingular matrices, more implementation details will be provided as we discuss the verification process.

Finally, we make a distinction between vanilla ACL2 and ACL2(r). Vanilla ACL2 numbers only include rationals and complex rationals. ACL2(r) is the version of ACL2 with support for real and complex numbers in general via non-standard analysis. In either case, computations on concrete numerics (rational, floating-point, or otherwise) are handled by the Common Lisp backend of ACL2 / ACL2(r), which enables the theorem prover to support native execution at modern speeds. In this paper, ACL2(r) is only necessary for taking square roots in the Cholesky decomposition algorithm. The square-root function used is the logical definition `acl2-sqrt`, which involves operations on nonstandard numbers. To make execution more amenable, we deploy a version of Cholesky which employs an iterative square-root function `sqrt-iter`, which has been verified to converge to `acl2-sqrt` [2], as a drop in replacement. It is possible to reason about square roots in vanilla ACL2 using only its algebraic properties, e.g., by augmenting the field of ACL2 numbers with some $\sqrt{}$. Instead of developing a new theory in ACL2, we decided to simply use ACL2(r).

IV. Linear Algebra Basics

One core idea of our approach is to recast algorithms in terms of operations on submatrices in a partitioned form of the original matrix. Originally, this partitioning was meant to make linear algebra code more intelligible and reasoning from a human perspective easier. However, it also enables machine reasoning in a verification context, which we will discuss in Sections V and VI. To see this partitioning in action, we derive the LU decomposition. An LU decomposition for a matrix A are matrices L and U where L is lower triangular with "1"s on the diagonal (i.e. unit lower triangular), U is upper triangular, and $A = LU$. In the interest of memory optimization, the unit lower triangular requirement makes it possible to overwrite the upper part of A with the upper part of U and the strictly lower part of A with the strictly lower part of L during the algorithm. Partition A, L, and U as follows:

$$A := \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}, \qquad L := \begin{pmatrix} 1 & \\ \ell_{21} & L_{22} \end{pmatrix}, \qquad U := \begin{pmatrix} \upsilon_{11} & u_{12}^T \\ & U_{22} \end{pmatrix} .$$

Before we continue, a note on notation: lower-case Greek letters are field scalars; lower-case Latin letters are vectors; upper-case Latin letters are matrices; and assume that any posed variables are "conformal", e.g., if A is $m \times n$, then a_{21} is $(m-1) \times 1$ and a_{12}^T is $1 \times (n-1)$. Setting $A = LU$ gives

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} 1 & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \upsilon_{11} & u_{12}^T \\ & U_{22} \end{pmatrix} . \quad (1)$$

We want Equation (1) to hold after performing the algorithm, i.e.

$$\alpha_{11} = \upsilon_{11} , \qquad a_{21} = \upsilon_{11}\ell_{21} ,$$
$$a_{12}^T = u_{12}^T , \qquad A_{22} = \ell_{21}u_{21}^T + L_{22}U_{22} .$$

Since A is given, u_{12}^T and υ_{11} are obvious. Solving for the remaining components of L and U forces

$$\ell_{21} = a_{21}\alpha_{11}^{-1} ,$$

and

$$L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T . \quad (2)$$

This suggests an algorithm which requires merely updating a_{21} and A_{22}. In particular, Equation (2) in the derivation above suggests a natural induction hypothesis which facilitates a recursive algorithm, i.e. our recursive call is to simply call the same LU decomposition algorithm on the smaller matrix $A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T$.

Consider our version of LU decomposition in Algorithm 2. Contrasting Algorithm 2 with Algorithm 1 elucidates the advantages of viewing numerical linear algebra algorithms through the lens of partitioned matrices. In terms of aesthetics alone, Algorithm 2 is more elegant than Algorithm 1. The technical advantages of this is that coherent code facilitates intelligent modularity, software reliability, codebase maintenance, and high performance, while enhancing confidence in its correctness.

Indeed, for our purposes, the major advantage of the presentation in Algorithm 2 is that partitioning the matrix at the start and end of the algorithm exposes an inductive invariant. At the start of the algorithm, all components of the matrix are highlighted red, indicating that none of the present components are in the desired "LU" form. The inductive invariant is that by the time a recursive call is initiated, all but the "bottom right" component is green, indicating that everything except A_{22} is already in LU form. To remedy the final form, Equation (2) indicates that we should simply call LU on A_{22}.

The progress of a right-greedy algorithm is visualized in Figure 1. Step (1) represents a matrix prior to the updates in a particular recursive call. Green indicates portions of the matrix that are already in the desired form and red indicates portions of the matrix that still need to be updated. Step (2) represents the matrix while updates are made during a recursive call. Purple indicates the portions of the matrix that are being updated. Step (3) represents the matrix just prior to the next recursive call. As the algorithm progresses, the portion of the matrix not yet in in the desired form reduces in size, until no part of the matrix needs to be updated, at which point the algorithm terminates.

What makes Algorithm 2 "right-greedy" is that the four bottom right purple-colored components as shown in Step (2) of Figure 1 are the submatrices of A to be updated within a recursive call.

The visualization of Figure 1 is algorithm agnostic in that the same progression holds for any right-greedy algorithm – not just LU. Indeed, we can undergo a similar derivation

Table 1 Common ACL2 functions, macros, and other commands used in this paper.

Command	*Description*
`defun`	Define a function symbol, e.g. `(defun add-1 (x) (+ x 1))`
`define`	A richer alternative to `defun`; enforces guard checking and more
`defthm`	Name and prove a theorem, e.g. `(defthm <-add-1 (< x (add-1 x)))`
`list`	Define a list, e.g. `(list 1 2 3)` returns `(1 2 3)`
`car`	Returns the head of a list, e.g. `(car (list 1 2 3))` returns `1`
`cons`	Construct a pair, e.g. `(cons 1 (list 2))` returns `(1 2)`
`/`	Divide two numbers or return the reciprocal of a number, e.g. `(/ 1 2)` or `(/ 2)`
`acl2-sqrt`	Square root of an ACL2 number, e.g. `(acl2-sqrt 2)`
`b*`	Binder for local variables; often used to simplify control flow statements

Table 2 ACL2 linear algebra functions.

Function	*Intended Signature*	*Description*
`matrixp`	$\mathbb{R}^{n\times m} \to \{\texttt{t},\texttt{nil}\}$	Matrix recognizer, e.g. `(matrixp (list (list 1 0 0)))` returns `t`
`m-emptyp`	$\mathbb{R}^{n\times m} \to \{\texttt{t},\texttt{nil}\}$	Empty matrix recognizer, e.g. `(m-emptyp nil)` returns `t`
`m-empty`	$\{\} \to \mathbb{R}^{0\times 0}$	Returns an empty matrix, e.g. `(m-empty)` returns `nil`
`mzero`	$\mathbb{N}\times\mathbb{N} \to \mathbb{R}^{n\times m}$	Returns a zero matrix, e.g. `(mzero 1 1)` returns `(list (list 0))`
`row-car`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{m}$	Returns the first row of a matrix
`col-car`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n}$	Returns the first column of a matrix
`row-cdr`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{(n-1)\times m}$	Remove a matrix's first row
`col-cdr`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times (m-1)}$	Remove a matrix's first column
`row-cons`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{(n+1)\times m}$	Append a row to a matrix
`col-cons`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times (m+1)}$	Append a column to a matrix
`m+`	$\mathbb{R}^{n\times m}\times\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Matrix addition
`m*`	$\mathbb{R}^{n\times m}\times\mathbb{R}^{m\times \ell} \to \mathbb{R}^{n\times \ell}$	Matrix multiplication
`sm*`	$\mathbb{R}\times\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Scalar-matrix multiplication
`sv*`	$\mathbb{R}\times\mathbb{R}^{n} \to \mathbb{R}^{n}$	Scalar-vector multiplication
`out-*`	$\mathbb{R}^{n}\times\mathbb{R}^{n} \to \mathbb{R}^{n\times n}$	Outer product multiplication
`get-L`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Get a matrix's lower triangular part
`get-U`	$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}$	Get a matrix's upper triangular part

Algorithm 2 LU decomposition (ACL2).

```
procedure LU($A \in \mathbb{R}^{m\times n}$)
    Partition $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$
    ▷ If $n, m > 1$, then $\alpha_{11} \in \mathbb{R}$, $a_{21} \in \mathbb{R}^{(n-1)\times 1}$, $a_{12}^T \in \mathbb{R}^{1\times(m-1)}$, $A_{22} \in \mathbb{R}^{(n-1)\times(m-1)}$
    if $m = 0$ or $n = 0$ then                          ▷ Edge case
        return ( )                                      ▷ Return an empty matrix
    else if $n = 1$ then                                ▷ Base case
        return $\begin{pmatrix} \alpha_{11} \\ a_{21}\alpha_{11}^{-1} \end{pmatrix}$
    else if $m = 1$ then                                ▷ Base case
        return $A$
    else                                                ▷ Recursive case
        $a_{21} := a_{21}\alpha_{11}^{-1}$
        $A_{22} := A_{22} - a_{21}a_{12}^T$
        return $\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & \text{LU}(A_{22}) \end{pmatrix}$
```

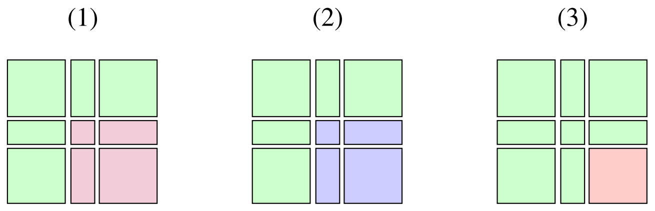


Figure 1: Progress of a right-greedy algorithm: (1) prior to updates ; (2) during updates; (3) after updates.

for the right-greedy Cholesky decomposition. Given a (real) symmetric positive definite matrix A, i.e. $A = A^T$ and $v^TAv > 0$ for all nonzero compatible vectors v, a Cholesky decomposition for A is a lower triangular matrix L such that $A = LL^T$. Partition as before:

$$A := \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}, \qquad L := \begin{pmatrix} \lambda_{11} & \\ \ell_{21} & L_{22} \end{pmatrix}.$$

Setting $A = LL^T$ gives

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \lambda_{11} & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \lambda_{11} & \ell_{21}^T \\ & L_{22}^T \end{pmatrix} \tag{3}$$

Algorithm 3 Cholesky decomposition (ACL2).

```
procedure CHOL($A \in \mathbb{R}^{m\times n}$)
    Partition $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$
      ▷ If $n, m > 1$, then $\alpha \in \mathbb{R}$, $a_{21} \in \mathbb{R}^{(n-1)\times 1}$,
        $a_{12}^T \in \mathbb{R}^{1\times(m-1)}$, $A_{22} \in \mathbb{R}^{(n-1)\times(m-1)}$
    if $m = 0$ or $n = 0$ then                         ▷ Edge case
        return ( )                                   ▷ Return an empty matrix
    else if $n = 1$ then                               ▷ Base case
        return $\begin{pmatrix} \sqrt{\alpha}_{11} \\ a_{21}\alpha_{11}^{-1} \end{pmatrix}$
    else if $m = 1$ then                               ▷ Base case
        return $\begin{pmatrix} \sqrt{\alpha}_{11} & a_{21}^T \end{pmatrix}$
    else                                              ▷ Recursive case
        $\alpha_{11} := \sqrt{\alpha_{11}}$
        $a_{21} := a_{21}\alpha_{11}^{-1}$
        $A_{22} := A_{22} - a_{21}a_{21}^T$
        return $\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & \text{CHOL}(A_{22}) \end{pmatrix}$
```

forces

$$\lambda_{11} = \pm\sqrt{\alpha_{11}}, \qquad \ell_{21} = a_{21}\lambda_{11}^{-1},$$
$$L_{22}L_{22}^T = A_{22} - \ell_{21}\ell_{21}^T. \tag{4}$$

For our purposes, we pick $\lambda_{11} = \sqrt{\alpha_{11}}$. Again, note that Equations (3) and (4) suggest a natural recursion. Our Cholesky decomposition algorithm is Algorithm 3.

Comparing Algorithm 3 with Algorithm 2 emphasizes the similar derivations, with the only contrast being the update to α_{11} in Algorithm 3. This extra update is necessary because the diagonals in a Cholesky decomposition are the same. In Algorithm 2, this update isn't necessary because we store the diagonal of an LU decomposition in L.

V. Verifying Right-greedy LU Decomposition

Here we describe our verification of an LU decomposition algorithm in ACL2. There are a few points to observe in this section. First, we describe some further ACL2 details as this will be the first instance of an ACL2 program in this paper. Second, note how we specify the algorithm's conditions for success. The textbook conditions require all principal leading submatrices to be nonsingular, which is a quantified statement and undesirable for executional efficiency. LU decomposition is only one numerical linear algebra algorithm; we are interested in verifying an entire *family* of algorithms. Third, the proof of correctness for our ACL2 LU decomposition program goes hand-in-hand with the derivation in Section IV. A pen-and-paper proof may directly apply the derivation as an induction step to prove the LU decomposition correct by construction. However, in ACL2, we first *specify* the LU decomposition algorithm as an executable program, and then prove it correct. Ideas in this section will be discussed at a higher level of abstraction in Section VII.

With the exception of some extra edge cases, Program 1 implements Algorithm 2 directly. The macro `define` is a wrapper for `defun` that simplifies many common aspects of function definition in ACL2, such as guards. Since ACL2 functions are total, guard checking is used to validate certain conditions or constraints before proceeding with execution. Guard checking is employed to enhance the robustness and reliability of ACL2 code by preventing the execution of code under inappropriate or unexpected circumstances.

The `b*` in the definition of `lu` is an example of an ACL2 macro for binding local variables with support for control flow. The first argument to `b*` is a list of "bindings" and the second argument is the ACL2 expression to which the bindings apply. For example, the binding `(alph (car (col-car A)))` declares the local variable `alph` to be equal to `(car (col-car A))`, i.e. the first element of the first column in `A`. If no early-exit bindings (such as `unless`) are triggered, then the value of the `b*` expression is the value of the second argument to `b*` with the bindings given by the first argument.

It is challenging to formalize the typical conditions for an LU decomposition of a matrix A to succeed. For one, they are presented as a quantified statement over the submatrices of A: all *principal leading submatrices of* A need to be nonsingular. If A is $n \times n$, the *principal leading submatrices of* A are the $k \times k$ "top left" submatrices of A, where $k \in [1, n]$. While ACL2 supports quantifiers via Skolem functions, these are not executable. We want a recognizer for LU decomposable matrices to be at least executable, not to mention efficient, because: (1) it can serve as a guard; and (2) our recognizer will also serve to induce an induction scheme for proving the correctness of `lu`. The other problem with the typical conditions is that nonsingularity is challenging to formalize. Thanks to the Invertible Matrix Theorem, there are over 20 equivalent characterizations for nonsingularity, most of which are computationally inefficient, require significant theory building, or also involve quantified statements.

Our solution is to use *Schur complements*. Consider Equation (1)

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} 1 & \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \upsilon_{11} & u_{12}^T \\ & U_{22} \end{pmatrix}. \tag{1 revisited}$$

Notice that if $\alpha_{11} \neq 0$, then A is LU decomposable iff the RHS of Equation (2)

$$L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T \tag{2 revisited}$$

is also LU decomposable. This is interesting because it reduces the condition for a matrix to be LU decomposable into a condition about a smaller matrix, which is reminiscent of some "induction step". Indeed, the RHS of Equation (2) is the *Schur complement of* α_{11} *in* A. While mathematical references commonly describe the conditions for a matrix to be LU decomposable in terms of the leading principal submatrices,

Program 1 ACL2 implementation of LU decomposition Algorithm 2.

```
(define lu ((A matrixp)) ...
 (b* (;; BASE CASES
      ((unless (matrixp A)) (m-empty))                  ;; If A not a matrix, return empty
      ((if (m-emptyp A)) A)                             ;; If A empty, return A
      (alph (car (col-car A)))                          ;; alph := "top left" scalar in A
      ((if (zerop alph))                                ;; If alph zero, return a zero
       (mzero (row-count A)                             ;;   matrix of the same dimensions
              (col-count A)))                           ;;   as A
      ((if (m-emptyp (col-cdr A)))                      ;; If A is a column, return
       (row-cons (list alph)                            ;;   [   1   ] [ a1 ] = [ a1 ] = A
                 (sm* (/ alph)                          ;;   [ a2/a1 ]          [ a2 ]
                      (row-cdr A))))                    ;;   [  ...  ]          [ ...]
      ((if (m-emptyp (row-cdr A))) A)                   ;; If A is a row, return A

      ;; PARTITION
      (a21 (col-car (row-cdr A)))                       ;; [ alph | a12 ] := A
      (a12 (row-car (col-cdr A)))                       ;; [ ---------- ]
      (A22 (col-cdr (row-cdr A)))                       ;; [ a21  | A22 ]

      ;; UPDATE
      (a21 (sv* (/ alph) a21))                          ;; a21 := a21 / alph
      (A22 (m+ A22 (sm* -1 (out-* a21 a12)))))          ;; A22 := A22 - a21 * a12

     ;; RECURSE                                         ;;  [ alph | a12     ]
     (row-cons (row-car A)                              ;;  [ -------------- ]
               (col-cons a21 (lu A22)))) ...)           ;;  [ a21  | LU(A22) ]
```

Program 2 ACL2 theorem for LU decomposition correctness.

```
(defthm lu-correctness
 (b* ((LU (lu A))
      (L  (get-unit-L LU))
      (U  (get-U LU)))
     (implies (and (equal (col-count A)
                          (row-count A))
                   (nonsingular-submatrices-p A))
              (equal (m* L U) A)))))
```

the proof that these conditions are sufficient reduces to an induction step that depends on Equation (2) [3].

To see the connection with nonsingular principal submatrices, observe that if no zeros appear after k recursive steps, then the k-th principal leading submatrix is nonsingular because its determinant is nonzero. Instead of reasoning with determinants, which are rarely useful in numerical algorithms [20], Schur complements provide a more concise ACL2 condition for success and generalizes to other algorithms.

VI. Verifying Right-greedy Cholesky Decomposition

Here we briefly describe our verification of the right-greedy Cholesky decomposition. Our focus will be on the commonalities with LU decomposition verification, some peculiarities in recognizing symmetric positive definite matrices, and less on ACL2 implementation details.

In order for a matrix A to have a Cholesky decomposition, it must be symmetric positive definite. Symmetric simply means $A^T = A$, but positive definite requires $v^T A v > 0$ for all nonzero compatible v. The latter is once again a quantified statement, which has all the implications discussed previously

Program 3 ACL2 theorem for Cholesky decomposition correctness.

```
(defthm chol-correctness
 (b* ((L       (get-L (chol A)))
      (Lt      (mtrans L)))
     (implies (and (equal (mtrans A) A)
                   (positive-definite-p A)
                   (equal (col-count A)
                          (row-count A)))
              (equal (m* L Lt) A)))))
```

in Section V. In order to define a executable recognizer for positive definite matrices, we once again look at Schur complements to satisfy *Sylvester's criterion* for a symmetric matrix to be positive definite. Sylvester's criterion states that a symmetric matrix is positive definite iff the principal leading submatrices are positive. The latter is equivalent to each principal leading submatrix having a positive determinant. From Section V, we saw that recursively computing Schur complements along the diagonal of A exhibits the determinants of the principal leading submatrices of A. Thus we merely need to check that each of these determinants are positive, which is how `positive-definite-p` in Program 3 is defined. The theorem for right-greedy Cholesky decomposition correctness then passes with minimal user-provided hints.

VII. Generalizing Rules for Automated Verification

Generalizing the ideas of Sections V and VI, our method to verifying the LU and Cholesky decompositions can be generalized to any right-greedy numerical linear algebra algorithms.

1) Define a recursive right-greedy algorithm.

2) Verify the derivation using the partitioned matrix approach.
3) Define a recursive recognizer for the appropriate class of matrices.
4) Induct according to a scheme automatically suggested by the recognizer.

The only steps which require human involvement is in Step 1 and 3. All that is required of a user is to define the algorithm to be verified and the class of matrices for which the algorithm computes. Step 4 is performed automatically because induction in ACL2 requires no human involvement. Step 2 is made automatic thanks to a formalized approach to deriving right-greedy algorithm. Observe that the RHS of both Equations (1) and (3) are simply instances of matrix multiplication between general partitioned matrices

$$\begin{aligned} BC &= \begin{pmatrix} \beta_{11} & b_{12}^T \\ b_{21} & B_{22} \end{pmatrix} \begin{pmatrix} \gamma_{11} & c_{12}^T \\ c_{21} & C_{22} \end{pmatrix} \\ &= \begin{pmatrix} \beta_{11}\gamma_{11} + b_{12}^T c_{21} & \beta_{11} c_{12}^T + b_{12}^T C_{22} \\ b_{21}\gamma_{11} + B_{22} c_{21} & b_{21} c_{12}^T + B_{22} C_{22} \end{pmatrix}. \end{aligned} \tag{5}$$

We formalize Equation (5) as an ACL2 rewrite rule which fires automatically when verifying the LU and Cholesky derivations. More generally, suppose we want to verify a right-greedy algorithm which computes B and C such that $BC = A$. The updates performed by a right-greedy algorithm's recursive step will be to compute β_{11}, γ_{11}, b_{21}, c_{21}, b_{12}^T, and c_{12}^T such that

$$\alpha_{11} = \beta_{11}\gamma_{11} + b_{12}^T c_{21}, \qquad a_{12}^T = \beta_{11} c_{12}^T + b_{12}^T C_{22},$$
$$a_{21} = b_{21}\gamma_{11} + B_{22} c_{21}$$

all hold. Then the algorithm's recursive call will be to find the decomposition

$$B_{22}C_{22} = A_{22} - b_{21} c_{12}^T. \tag{6}$$

The above identities are easily translated into ACL2 rewrite rules as an instantiation of the rewrite rule for Equation (5). Given these rewrite rules, the induction in Step 4 discharges automatically.

The LU and Cholesky decompositions we verify are instantiations of the above. Note that Equation (2)

$$L_{22}U_{22} = A_{22} - a_{21}\alpha_{11}^{-1} a_{12}^T \tag{2 revisited}$$

is a case of Equation (6). If $\alpha_{11} \neq 0$ also holds, then $A = LU$. Similarly, Equation (4)

$$L_{22}L_{22}^T = A_{22} - \ell_{21}\ell_{21}^T \tag{4 revisited}$$

is a case of Equation (6). If $\alpha_{11} > 0$ and $a_{12} = a_{21}$ also hold, then $A = LL^T$. These rules follow directly from Equation (5) with little user-guidance in ACL2.

VIII. Conclusion

We demonstrated a formal method for automatically verifying right-greedy numerical linear algebra algorithms. At the heart of our approach is the partitioned matrix environment which we use to define and verify derivations of recursive right-greedy algorithms. Partitioning and defining algorithms in this manner promotes automated reasoning and verification by introducing induction schemes. We've implemented our method using the ACL2 theorem prover. The choice of theorem prover is not vital provided that it supports induction. However, ACL2 provides two additional major benefits. First ACL2 offers a high degree of automation beyond what is possible with other theorem provers. Second, our verified formalizations are natively executable within the logic of ACL2; this provides industrial-level computational performance. This is particularly important because numerical algorithms are usually meant to be implemented and executed in real world systems. No other theorem prover offers the same level of execution performance.

Our work involved writing 1593 lines of new ACL2 code, used to introduce 262 new ACL2 events. Verifying the new ACL2 code required 6 793 576 prover steps, which were performed automatically. Performing these steps took 9.76 seconds and ACL2 used 1.37 GB of memory on a laptop. The interested reader may try using our code [21] to decompose their own matrices.

There are immediate applications for our work. We discussed determinants in Sections V and VI. Note that if $A = LU$ is LU decomposable, then $\det(A) = \det(L)\det(U) = \det(U)$ is simply the product of the diagonal of U. Another consequence of formalizing a right-greedy LU decomposition algorithm is that the computed U is actually the row echelon form of A. This means that `(defun ge (A) (get-U (lu A)))` is the ACL2 verified formalization of Gaussian elimination. One very important application of LU decomposition is that it can be used to solve a linear system $b = Ax$. If $A = LU$, then $b = Ax = (LU)x = L(Ux)$ indicates that one can first solve $b = Ly$ via forwards substitution and then $y = Ux$ via backwards substiution to solve $b = Ax$. Cholesky can be applied similarly. We have formalized backwards and forwards substitution in ACL2, which is beyond the scope of this paper, but this indicates we have a verified and executable method for solving systems of linear equations in ACL2.

The applications of numerical linear algebra in which safety, correctness, and accuracy are critical indicates a need for formally verified numerical linear algebra software systems. In addition to solving linear systems, we can pursue the verification of other executable numerical linear algebra algorithms. The class of right-greedy algorithms includes classical QR decomposition, which has yet to be formally verified. Proving this in ACL2 would give rise to verified executable implementations of LU, Cholesky, and QR decompositions (sometimes referred to as the "three amigos" by the scientific computing community), which could serve as the beginnings of a fully verified numerical linear algebra library.

Right-greedy algorithms are a major player in scientific and high-performance computing, with many dozens of such algorithms serving as the basis for ongoing research. Targeting improved performance on not just dense, but also sparse and block matrices across architectures such as GPUs and FGPAs place variants of right-looking algorithms in the hundreds. Other names for "right-greedy" are "right-looking", "data-driven" or "submatrix". Our approach can be augmented to verify other families of numerical linear algebra algorithms. In the FLAME framework, identifying different loop invariants suggests derivations of other algorithmic flavors, such as "left-greedy", "up-greedy", "bordered", etc. We want to develop formal methods for automatically verifying these other families of numerical linear algorithms.

Another important FLAME idea is using the partitioned matrix approach to perform backwards error analysis. Formalizing bounds on errors and proving the convergence of iterative numerical algorithms are vital to the reliability of their implementations. This would involve notions such as matrix norms or the condition number of a matrix. ACL2 supports matrix and vector analysis by way of nonstandard analysis [11], [22] and recent developments in the ACL2 research community include a deep embedding of floating-point numbers into the ACL2 logic. This provides all the formal tools necessary to perform ACL2 backwards error analysis of numerical linear algebra algorithms and we intend to pursue these sorts of proofs.

Linear algebra underlies modern scientific computing infrastructure. It is critical real world linear algebra computations are accurate and correct. We endeavour to guarantee the veracity of these computations by developing verified numerical linear algebra libraries. Our method for automating the verification of right-greedy numerical linear algebra algorithms is foundational to achieving this objective.

Acknowledgments

We thank Robert van de Geijn, Margaret Myers, and the anonymous reviewers for their helpful comments and feedback.

References

[1] M. Kaufmann and J. S. Moore, "ACL2 home page," https://cs.utexas.edu/~moore/acl2/, 1997, accessed 2024-06-25.

[2] R. A. Gamboa and M. Kaufmann, "Nonstandard analysis in ACL2," *J. Autom. Reason.*, vol. 27, no. 4, p. 323–351, November 2001.

[3] G. W. Stewart, *Matrix Algorithms Volume I: Basic Decompositions*, 1st ed., 1998.

[4] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal linear algebra methods environment," *ACM Trans. Math. Softw.*, vol. 27, no. 4, pp. 422–455, December 2001.

[5] C. Kwan, "Classical LU decomposition in ACL2," *Electronic Proceedings in Theoretical Computer Science*, vol. 393, pp. 1–5, November 2023.

[6] L. Lambán, F. J. Martín-Mateos, J. Rubio, and J.-L. Ruiz-Reina, "Using abstract stobjs in ACL2 to compute matrix normal forms," in *Interactive Theorem Proving*, M. Ayala-Rincón and C. A. Muñoz, Eds. Cham: Springer International Publishing, 2017, pp. 354–370.

[7] J. Hendrix, "Matrices in ACL2," 2003. [Online]. Available: https://www.cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/hendrix/hendrix.pdf

[8] R. Gamboa, J. Cowles, and J. Van Baalen, "Using ACL2 arrays to formalize matrix algebra," 2003. [Online]. Available: https://www.cs.uwyo.edu/~ruben/static/pdf/matalg.pdf

[9] C. Kwan and M. R. Greenstreet, "Real vector spaces and the Cauchy-Schwarz inequality in ACL2(r)," *Electronic Proceedings in Theoretical Computer Science*, vol. 280, pp. 111–127, October 2018.

[10] C. Kwan, Y. Peng, and M. R. Greenstreet, "Cauchy-Schwarz in ACL2(r) abstract vector spaces," *Electronic Proceedings in Theoretical Computer Science*, vol. 327, pp. 90–92, May 2020.

[11] C. Kwan and M. R. Greenstreet, "Convex functions in ACL2(r)," *Electronic Proceedings in Theoretical Computer Science*, vol. 280, pp. 128–142, October 2018.

[12] W. A. Hunt, Jr., V. Ramanathan, and J. S. Moore, "VWSIM: A circuit simulator," in Proceedings Seventeenth International Workshop on the *ACL2 Theorem Prover and its Applications,* Austin, Texas, USA, 26th-27th May 2022, ser. Electronic Proceedings in Theoretical Computer Science, R. Sumners and C. Chau, Eds., vol. 359. Open Publishing Association, 2022, pp. 61–75.

[13] Z. Shi and G. Chen, "Integration of multiple formal matrix models in Coq," in *Dependable Software Engineering. Theories, Tools, and Applications*, W. Dong and J.-P. Talpin, Eds. Cham: Springer Nature Switzerland, 2022, pp. 169–186.

[14] "Lean mathlib3 documentation: LDL decomposition," https://leanprover-community.github.io/mathlib_docs/linear_algebra/matrix/ldl.html, accessed 2023-07-13.

[15] R. Thiemann and A. Yamada, "Matrices, Jordan normal forms, and spectral radius theory," *Archive of Formal Proofs*, August 2015, https://isa-afp.org/entries/Jordan_Normal_Form.html, Formal proof development.

[16] Z. Shi, Y. Zhang, Z. Liu, X. Kang, Y. Guan, J. Zhang, and X. Song, "Formalization of matrix theory in HOL4," *Advances in Mechanical Engineering*, vol. 6, pp. 195–276, 2014.

[17] P. Bientinesi and R. A. van de Geijn, "Goal-oriented and modular stability analysis," *SIAM J. Matrix Anal. Appl.*, vol. 32, no. 1, p. 286–308, March 2011.

[18] W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of the Royal Society of London Series A*, vol. 375, no. 2104, September 2017.

[19] ACL2, *User manual for the ACL2 Theorem Prover and the ACL2 Community Books*, accessed 2024-07-12. [Online]. Available: https://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html

[20] L. N. Trefethen and D. Bau, III, *Numerical Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[21] M. Kaufmann and J. S. Moore, "ACL2 system and community books," https://github.com/acl2/acl2, 2014.

[22] C. Kwan, "Towards formalized matrix analysis and algorithms," in *International Symposium on Artificial Intelligence and Mathematics*, 2022.

Formally Verified Rounding Errors of the Logarithm-Sum-Exponential Function

Paul Bonnot*, Benoît Boyer†, Florian Faissole†, Claude Marché* and Raphaël Rieu-Helft‡
*Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France
†Mitsubishi Electric R&D Centre Europe, Rennes, France
‡TrustInSoft, 75014 Paris, France

***Abstract*—We study the numerical accuracy of some specific computer program performing numerical computations. Such a numerical accuracy is expressed in terms of a bound on the difference between the floating-point computation and the corresponding rounding-free computation using mathematical real numbers. We do not only seek to discover such a bound "on paper" but we aim at obtaining computer-assisted formal proofs that this bound is correct for any possible inputs. The function we study comes from the domain of machine learning: a function computing the logarithm of the sum of exponentials of a sequence. The bound obtained is an original result, parameterized by the error bounds of the underlying implementations of the logarithm and exponential functions. The methodology we follow to conduct our formal proofs is also original, using a combination of the Why3 environment for deductive verification, an original modelling of floating-point computations using *unbounded* numbers, and the J^3 environment for proving properties on C source code.**

I. Introduction

Software is involved in many industrial systems nowadays. In cyber-physical systems in a broad sense, the software controlling a system must perform numerical computations, that are typically based on floating-point arithmetic. The floating-point representation of numbers, and the operations on them, are standardized by the IEEE-754 standard [43]. Despite of these rules, guessing the accuracy of a program that compounds thousands of elementary operations, without software assistance, becomes almost impossible. However, recent history has shown that underestimating these errors could have catastrophic consequences [57]. That explains the recent interest for the formal verification of floating-point properties of numerical programs in proof assistants [12], [55] or deductive verification platforms [14]. Formally proving the accuracy of floating-point computations is a complex topic addressed by different approaches in the scientific literature. Recent overviews of this topic can be found in the Handbook of Floating-Point Arithmetic [53], or surveys by Melquiond [50] and Boldo *et al.* [18].

In contrast with elementary operations, the accuracy of mathematical functions, say as provided by the `libm` library for C code, is not enforced by the standard and could depend on the target architecture and on a specific library. The same goes for more complex mathematical algorithms, for which the accuracy strongly depends on the accuracies of underlying `libm` functions. In addition, the proof makes sense only if the parameterized bound assumed on `libm` functions can be realized by some implementation. For these reasons, there was very little effort to formally verify programs that call mathematical functions implementations.

An example of algorithm making use of `libm` functions is the *Log-Sum-Exp* algorithm, abbreviated as LSE. It is typically used in machine-learning applications [24], [35], [52]. This algorithm is a smooth approximation of the max function, the smoothness property of inner functions being a pre-requisite for the efficiency of machine-learning algorithms. It applies to an n-dimensional vector $a = (a_1, \ldots, a_n)$ and computes the logarithm of the sum of exponentials over its a_i components:

$$\mathsf{LSE}(a) = \log\left(\sum_{1\le i\le n} \exp(a_i)\right)$$

This function is frequently used in implementations of statistical classifiers [42], [48] and satisfies the following property:

$$\max_{1\le i\le n}(a_i) \le \mathsf{LSE}(a) \le \max_{1\le i\le n}(a_i) + \log(n)$$

Blanchard *et al.* [10] showed that the rounding error of a floating-point implementation of the LSE function can be bounded by relatively tight values as long as no overflow or underflow occurs. They present a pen-and-paper proof taking advantage of both floating-point arithmetic specificities and mathematical properties of the log and exp functions. They assume that the implementations of these two functions are correctly rounded, that is, their relative errors are bounded by the ε unit round-off (see Section II). Therefore, they provide the best accuracy that can be achieved in the considered floating-point format.

Compared to Blanchard *et al.* [10], one of the contribution of this paper is to consider the possibility of using less accurate but more efficient implementations of exp and log. Such implementations may be useful in contexts involving energy-saving small devices like IoT [38]. Another contribution is to give formal proofs of our results. Generally speaking, obtaining formal proofs on the accuracy of floating-point programs is not a simple task. One can start from a software environment for proving functional properties of programs, and augment it with a formalization of floating-point arithmetic, typically via a library built on top of a formalization of real numbers

[0]This research was supported partly by the bilateral contract ProofInUse-MERCE and partly by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030"

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_31

that provides a rounding function and its logical properties. It was done for example by Boldo and Filliâtre [17] using the Coq proof assistant, augmented with the Flocq library [22] for floating-point arithmetic, allowing to prove programs using the Coq general-purpose environment. These proofs typically require a large amount of manually written proof steps. To obtain a higher degree of automation, Ayad and Marché [5] proposed a setting making use of the Frama-C [46] environment for static analysis on C source code, with a dedicated library of ACSL [9] specifications that allows to discharge the proofs to various theorem provers, in particular SMT solvers. Boldo and Marché [19] presented an overview of what could be achieved on increasingly complex codes, using combination of automated solvers and the Coq proof assistant for the most complex proof obligations. With the addition, later on, of some built-in support for floating-point operations in SMT solvers, Fumex *et al.* [34] showed that this methodology can reach a fairly high amount of automation.

Initially we planned to follow the methodology above to prove a C code for LSE. Yet, to achieve the proofs in a reasonably simple manner, we achieved two important points that should be emphasized: first, the use of the intermediate language WhyML, and second the use *unbounded* floating-point numbers. The rest of this paper is organized as follows. In Section II we expose original results bounding the accuracy of LSE, parameterized by assumed accuracies of implementations of $\exp$ and $\log$. Unbounded Floating-Point numbers are detailed in Section II-B. We present our formalization in Section III, formalizing our results up to a formal proof of a C code computing LSE. The WhyML language is introduced in Section III. We discuss related work in Section IV and conclude in Section V with an overview of future work. Due to lack of space, we do not include pen-and-paper proofs here: these proofs can be found in an extended research report of ours [23], to which the reader should refer for any more technical details. The code formalizing our results is publicly available on the Toccata gallery [20], specifically at URL https://toccata.gitlabpages.inria.fr/toccata/gallery/lse.en.html.

II. Statement of Accuracy Results

A. Preliminaries on Floating-Point Arithmetic

The IEEE-754 standard [43] defines several formats of representation of floating-point numbers. A format is characterized by a precision p as well as upper and lower bounds $e_{\max}$ and $e_{\min}$ for the exponent. A floating-point number is either a value among $+\infty$, $-\infty$ and NaN or a value $\pm m \times 2^{e-p+1}$ where $m, e \in \mathbb{Z}$, $0 \leq m \leq 2^p - 1$ and $e_{\min} \leq e \leq e_{\max}$. The largest representable number in this format is $\mathsf{maxf} = (2 - 2^{-p-1}) \times 2^{e_{\max}}$, and the smallest positive representable number is $2^{e_{\min}-p+1}$.

In this paper we are not interested in a particular format since our proof methodology is independent of the format used, however only two formats are currently supported in our formal proofs: single format (32 bits) where $p = 24$, $e_{\max} = 127$ and $e_{\min} = -126$, and double format (64 bits) where $p = 53$, $e_{\max} = 1023$ and $e_{\min} = -1022$. For simplicity we focus on the double format.

We use the symbol rnd to denote the *rounding* of a real number to a floating-point number. The IEEE-754 standard defines several rounding modes. Here we consider only the mode *nearest-ties-to-even*: when a real number x lies within an interval $[x_1; x_2]$ of two consecutive floating-point numbers, then $\mathsf{rnd}(x)$ is either x_1 or x_2: the one of these which is closest to x, or in case x is exactly in the middle, the one among x_1 and x_2 whose mantissa is even. Also, when x is too large (larger than or equal to the middle of maxf and $2 \times 2^{e_{\max}}$), $\mathsf{rnd}(x)$ is $+\infty$.

We use the symbols $\oplus$, $\ominus$, $\otimes$, $\oslash$ to denote the basic operations of addition, subtraction, multiplication and division of floating-point numbers. As specified by IEEE-754, all these operations must use the best possible rounding, that is $x \oplus y = \mathsf{rnd}(x+y)$ and similarly for the three other operations.

The main property of the rounding function that we use in this paper is the following: for any real number x such that $|x| \leq \mathsf{maxf}$, $\mathsf{rnd}(x)$ is finite and

$$|\mathsf{rnd}(x) - x| \leq \varepsilon|x| + \eta \qquad (1)$$

where $\varepsilon = \frac{2^{-p}}{1+2^{-p}}$ and $\eta = 2^{e_{\min}-p}$. This property can be considered as well-known and folklore in the literature, see for example Jeannerod and Rump [44]. In seminal publications, such as the Handbook of Computer Arithmetic [54] or Higham's survey [41], the simpler term $\varepsilon = 2^{-p}$ is used instead of $\frac{2^{-p}}{1+2^{-p}}$, inducing a slightly larger bound. Jeannerod and Rump [44, Theorem 2.1] showed that the refined bound is actually optimal in the sense that there exist some inputs values and floating-point formats (with certain conditions) for which it is attained. In most cases, the precision gain obtained using this optimal bound instead of 2^{-p} is small. Anyway, the latter results and proofs simply use the symbol ε to denote either of the bounds.

As remarked by Jeannerod and Rump [44], Property (1) can be refined in the special case of addition because underflowing additions are exact:

$$|(x \oplus y) - (x + y)| \leq \varepsilon|x + y| \qquad (2)$$

that is, the term η can be removed from Formula (1). Moreover, it should be noted that (see for example the Handbook [54])

$$|(x \oplus y) - (x + y)| \leq |x| \qquad (3)$$

and symmetrically

$$|(x \oplus y) - (x + y)| \leq |y| \qquad (4)$$

The combination of the formulas (2), (3) and (4) is used later on to obtain bounds on compound sums.

B. Unbounded Floating-Point Numbers

The notion of unbounded floating-point number is somewhat simple, and is in fact not original: it is commonly used in the literature on numerical programs [54] and also in advanced formalization such as Flocq [22]. Roughly speaking, unbounded floating-point number are very much like standard IEEE floating-point numbers, except that their exponent can be arbitrarily large: it is a value $\pm m \times 2^{e-p+1}$ where $0 \leq m \leq 2^p - 1$, $e_{\min} \leq e$ without upper bound on e. As a consequence, unlike standard floating-point numbers, the four basic operations on unbounded floating-point numbers *never overflow*. There is no need for special values for infinities to represent the result of unbounded floating-point operations. On the other hand, notice that unbounded floats include sub-normal numbers. There is an injection from finite IEEE float numbers to unbounded float numbers. The properties (2), (3) and (4) indeed hold for unbounded floating-point numbers. This fact allows us to separate the proofs concerning functional behavior of numerical programs from the proof of absence of overflow: to prove a property on floating-numbers it suffices to prove the same on unbounded floats, and separately prove that each floating-point operation involved does not overflow nor produces NaN values.

C. Accuracy of Compound Summations

The compound sum of a vector $(a_1, \ldots, a_n)$ of floating-point numbers is the sum of all a_i. Defining it properly is more complex than the compound sum of real numbers because $\oplus$ is not associative. It is thus necessary to choose the order in which the additions are done. We make the choice to associate to the left, meaning that we define the compound sum from a_m (included) to a_k (excluded), denoted by $\bigoplus_{m \leq i < k} a_i$, by the following recursive equations.

$$\begin{aligned} \bigoplus_{m\leq i<k} a_i &= 0 \qquad \text{if } k \leq m \\ \bigoplus_{m\leq i<k} a_i &= \left(\bigoplus_{m\leq i<k-1} a_i \right) \oplus a_{k-1} \qquad \text{when } m < k \end{aligned}$$

Associating to the left is important because it impacts the final result of a sum. Yet the bounds we prove in the following are invariant by permutation of the element of the input vector. In other words, the same bounds could be proved when the sum is performed in any other order.

The following states a bound on compound sums, as a slight reformulation of a theorem by Jeannerod and Rump [44].

Theorem II.1 (Accuracy of compound sums)**.** *For any vector a of unbounded doubles, and any $m \leq n$:*

$$\left| \bigoplus_{m\leq i<n} a_i - \sum_{m\leq i<n} a_i \right| \leq (n-m-1)\varepsilon \sum_{m\leq i<n} |a_i|$$

The proof of this theorem [23] is far for trivial and make a clever use of properties (2), (3) and (4). From the previous theorem, we deduce the following corollary, which is useful in the proofs we perform.

Corollary II.2 (Bound on sums)**.** *For any constant S and M_a, any vector a, any indices m and n such that $n - m \leq S$ and $|a_i| \leq M_a$ for any $m \leq i < n$ we have*

$$\left| \bigoplus_{m\leq i<n} a_i \right| \leq M_a \times S \times (1 + \varepsilon(S-1))$$

D. Approximations of exp *and* log

In implementations using floating-point numbers, not only the sum is subject to rounding, but also the computations of functions exp and log. Here, we do not discuss any particular implementations of these two functions. Instead, we assume given implementations for them with given bounds in the rounding errors they perform. We do not want to rely, as Blanchard *et al.* [10] do, on perfectly rounded implementations of exponential and logarithm. Instead we assume we have implementations that are possibly less precise, the precision of them being specified as parameters.

Concerning exponential first, we assume given an implementation $\widehat{\exp}$ which satisfies the following property: for any real x such that $|x| \leq M_{\exp}$,

$$|\widehat{\exp}(x) - \exp(x)| \leq E_{\exp} \exp(x)$$

where $M_{\exp}$ and $E_{\exp}$ are two positive parameters. In the following we need to assume $E_{\exp} \leq 0.5$, a reasonable assumption, which in particular implies that $\widehat{\exp}(x)$ is always non-negative. Concerning logarithm we assume similarly an implementation satisfying the following property: for any real x such $0 < x \leq M_{\log}$

$$|\widehat{\log}(x) - \log(x)| \leq E_{\log} |\log(x)|$$

where $M_{\log}$ and $E_{\log}$ are positive parameters.

Notice that we do not claim that there exist implementations of approximations of exponential and logarithm, satisfying the properties above, for any value of the parameters $E_{\exp}$, $M_{\exp}$, $E_{\log}$ and $M_{\log}$. We just assume we are given some. Indeed it is known in the literature that such implementations exist for double precision, with a correct rounding, that is with $E_{\exp} = E_{\log} = \varepsilon$, $M_{\log} = \mathsf{maxf}$, and $M_{\exp}$ at most 708 (for larger values the exponential overflows): see for example Daramy *et al.* [29] and the implementations provided by the CORE-MATH project [56].

Notice also that we assume only some relative error ($E_{\exp}$ and $E_{\log}$) but no absolute error. For exponential, this is not needed because for an argument at least -708 the result is never a sub-normal. For a completely different reason, the logarithm do not need to return any sub-normal either, because the logarithm of the floating-point successor of 1 is around 2^{-52}, larger than a sub-normal too (and similar for the predecessor).

E. Accuracy of LSE

Our main result concerning the accuracy of the computation of $\widehat{\mathsf{LSE}}$ is given by Theorem II.5 below. To prove this theorem we need to establish first a few auxiliary lemmas. In these lemmas, we consider arbitrary positive constants A and B.

The first lemma is in fact a generalization of Theorem II.1 on the accuracy of compounds sums, when the input vector is itself subject to errors.

Lemma II.3 (Accuracy of sums, generalized)**.** *Given any vectors a and $\widehat{a}$ such that for all i, $|\widehat{a}_i - a_i| \leq A|a_i| + B$ we have:*

$$\left| \bigoplus_{0\leq i<n} \widehat{a}_i - \sum_{0\leq i<n} a_i \right| \leq (A + (n-1)\varepsilon(1+A)) \sum_{m\leq i<n} |a_i| + Bn\,(1+(n-1)\varepsilon)$$

The next lemma is necessary to propagate errors bounds through the mathematical log.

Lemma II.4 (Error propagation for mathematical logarithm)**.** *For any positive real numbers x and $\widehat{x}$ such that $|\widehat{x} - x| \leq Ax$, with $A < 1$ we have:*

$$|\log \widehat{x} - \log x| \leq -\log(1-A)$$

These results are combined to get the final result we target, as follows.

Theorem II.5 (Accuracy of LSE)**.** *For any $n \geq 1$, and no larger than 2^{51}, any vector a of size n such that for all i, $|a_i| \leq M_a$ for some $M_a \leq M_{\exp}$, and assuming that*

$$\exp(M_a)(1+E_{\exp})n(1+\varepsilon(n-1)) \leq M_{\log} \tag{5}$$

we have

$$\left|\widehat{\mathsf{LSE}}(a) - \mathsf{LSE}(a)\right| \leq E_{\log}|\mathsf{LSE}(a)| - \log\left(1 - (E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\right)(1+E_{\log})$$

The hypothesis (5) above is required to call the $\widehat{\log}$ function on the proper interval of definition. A bound on the size of the input is needed to apply Lemma II.4 with $A = E_{\exp} + (n-1)\varepsilon(1+E_{\exp})$: to show that A is smaller than 1, together with the hypothesis $E_{\exp} \leq 0.5$, the bound 2^{51} on n suffices.

F. Discussion on the Variations of the Bound on Accuracy

The error bound of $\widehat{\mathsf{LSE}}$ has two parts :

- A relative part, which is $E_{\log}$
- A constant part :

  $$-\log\left(1 - (E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\right)(1+E_{\log})$$

  We note that $-\log(1-x) < 2x$ for $x \leq \frac{1}{2}$. We can therefore bound the constant error by

  $$2 \times (E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\,(1+E_{\log})$$

  The factors that dominate the constant bound are $E_{\exp}$ and $\varepsilon \times (n-1)$.

In Section IV, we compare this bound with the one proposed by Blanchard *et al.* [10].

The error bound grows linearly with $E_{\log}$, $E_{\exp}$ and n. Since it is possible to choose an implementation of $\widehat{\log}$ and $\widehat{\exp}$ with specific bounds, having the error bound of $\widehat{\mathsf{LSE}}$ depending on $E_{\exp}$ and $E_{\log}$ is useful in order to control the error.

To give some instances of the obtained bound, we can choose specific values for the parameters $E_{\log}$, $E_{\exp}$, $M_{\exp}$, $M_{\log}$, M_a and n. Let us assume reasonable bounds in practice on n and M_a that is $2^{10} = 1024$ and $M_a = 25$.

- Let us assume first we have some correctly rounded implementations of exponential and logarithm, that is $E_{\exp} = 2^{-53}$ and $E_{\log} = 2^{-53}$. Then, to ensure that hypothesis (5) of Theorem II.5 holds, it suffices to have $M_{\log}$ larger than

  $$\begin{aligned} &\exp(M_a)(1+E_{\exp})n(1+\varepsilon(n-1)) \\ &\leq \exp(25)(1+2^{-53})2^{10}(1+2^{-53}\times 1023) \\ &\leq 7.38\times 10^{13} \end{aligned}$$

  Assuming thus that the implementation of $\widehat{\log}$ is correctly rounded on the domain given by the bound $M_{\log}$ above, the relative error on LSE is $E_{\log} = 2^{-53}$ and the absolute error is bounded by

  $$\begin{aligned} &2\times(E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\,(1+E_{\log}) \\ &\leq 2\times\left(2^{-53} + 1023\times 2^{-53}(1+2^{53})\right)(1+2^{-53}) \\ &\leq 2.28\times 10^{-13} \end{aligned}$$

- Let us assume less precise implementations of exponential and logarithm with $E_{\exp} = 2^{-40}$ and $E_{\log} = 2^{-36}$. These are some bounds for efficient implementations of exponential and logarithm that empirically seemed sufficiently accurate and energy-saving for industrial applications like IoT systems or deep-learning frameworks [38]. Then, to ensure that hypothesis (5) of Theorem II.5 holds, it suffices to have $M_{\log}$ larger than

  $$\begin{aligned} &\exp(M_a)(1+E_{\exp})n(1+\varepsilon(n-1)) \\ &\leq \exp(25)(1+2^{-40})2^{10}(1+2^{-53}\times 1023) \\ &\leq 7.38\times 10^{13} \end{aligned}$$

  that is roughly the same bound as above with correct rounding on $\widehat{\exp}$ and $\widehat{\log}$. In other words, the required bound on the input domain of logarithm depends mostly on the bound on inputs and the number of elements in the input sequence. The relative error on the computation of LSE is now $E_{\log} = 2^{-36}$ and the absolute error is bounded by

  $$\begin{aligned} &2\times(E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\,(1+E_{\log}) \\ &\leq 2\times\left(2^{-40} + 1023\times 2^{-53}(1+2^{53})\right)(1+2^{-36}) \\ &\leq 2.05\times 10^{-12} \end{aligned}$$

  This absolute error is roughly twice the absolute error of the case with correct rounded implementations of $\exp$ and $\log$.

Notice that if the size of the sequence is significantly larger than the assumed bound 1024, then the required value for $M_{\log}$ gets significantly larger, indeed it increases roughly linearly with this size.

```
(**  The type of unbounded floats in "double" format *)
type udouble

(** injection of udouble to real numbers *)
function to_real udouble : real

(** The rounding function *)
function uround mode real : udouble

constant eps:real = 0x1p-53 / (1.0 + 0x1p-53)
constant eta:real = 0x1p-1075

axiom uround_rne: forall x:real.
  abs (uround RNE x - to_real x) <= eps * abs x + eta

(** addition *)
function uadd (x y:udouble) : udouble =
  uround RNE (to_real x + to_real y)

(** properties (2), (3) and (4) *)
axiom add_rounding : forall x y:udouble.
  abs (to_real (uadd x y) - (to_real x + to_real y))
    <= abs (to_real x + to_real y) * eps

axiom add_bound_left: forall x y:udouble.
  abs (to_real (uadd x y) - (to_real x + to_real y))
    <= abs (to_real x)

axiom add_bound_right: forall x y:udouble.
  abs (to_real (uadd x y) - (to_real x + to_real y))
    <= abs (to_real y)
```

Fig. 1. Theory of unbounded doubles in WhyML (excerpt): conversion to real numbers, rounding, addition and its properties.

```
constant exp_max_value :real  (* constant M_exp *)
axiom exp_max_value_spec: 0.0 < exp_max_value

constant exp_error:real  (* constant E_exp *)
axiom exp_error_bound : 0.0 < exp_error <= 0.5

function u_exp (x:udouble) : udouble  (* function exp^ *)
axiom u_exp_spec : forall x:udouble.
  abs (to_real x) <= exp_max_value →
    abs (to_real (u_exp x) - exp (to_real x))
      <= exp (to_real x) * exp_error
```

Fig. 2. Declaration of $\widehat{\exp}$ in WhyML, with assumed accuracy.

```
1  let lemma lse_accuracy (a:int → udouble) (size:int) (max_a:real)
2    requires { 1 <= size }
3    requires { from_int (size - 1) <= 0x1p51 }
4    requires {
5      forall i. 0 <= i < size →
6        abs (to_real (a i)) <= max_a <= exp_max_value }
7    requires {
8      exp max_a * (1.0 + exp_error) *
9        from_int size * (1.0 + eps * from_int (size - 1))
10     <= log_max_value }
11   ensures {
12     let err = exp_error + eps * from_int (size - 1)
13               * (1.0 + exp_error) in
14     abs (to_real (u_lse a size) - lse_exact a size) <=
15       log_error * abs (lse_exact a size)
16       - log (1.0 - err) * (1.0 + log_error) }
```

Fig. 3. Statement of Theorem II.5 in WhyML.

III. Formalization of the Accuracy Result

In this section we show how we formalized the statement of Theorem II.5. We first summarize the methodology we followed. We then consider successively the formalization of LSE accuracy theorem in WhyML (Section III-B) and then a proof of a corresponding C code (Section III-C).

Our methodology makes use of the unbounded floating-point numbers introduced in Section II-B and heavily relies on WhyML. WhyML is the language of Why3 [11], a general-purpose environment for deductive verification. Why3 is used as an intermediate tool by several front-ends including Frama-C [46] for C code and by Spark for Ada code [49]. The Why3 environment allows the user to access a large set of different provers, including Coq and Gappa. Moreover, nowadays there are alternatives to the use of Coq for proving pure mathematical facts, including dReal [36] and Metitarski [1]. Concerning the reasoning on floating-point computation, Why3 gives access to SMT solver which support the SMT-LIB floating-point theory, such as CVC4 [7], cvc5 [6], Z3 [33] and Alt-Ergo-FPA [26]. But WhyML also proposes to the user a large set of techniques and tools to achieve complex proofs, for example via the use of *lemma functions*, which are, roughly speaking, a way to construct a proof by writing a program.

A. WhyML Formalization of Unbounded Floating-Point Numbers

A starting point of our formalization was to design a new WhyML theory for unbounded floats. An excerpt of that theory is given in Figure 1. In this theory, the type `udouble` is abstract, and only assumed to be given a function `to_real` which returns the real number represented by any `udouble`. The rounding function `uround` that rounds any real number to a `udouble` is also declared abstractly. The basic operations are defined as the rounding of the real operations. The properties (2), (3) and (4) are stated as axioms in the theory. To provide guarantees that this theory is consistent with IEEE floats, it is *realized* using Coq and its Flocq library.

B. Accuracy of LSE proved in WhyML

To start with, we need to declare the approximations of exp and log that we consider. The declaration of $\widehat{\exp}$ is shown in Figure 2 and the one of $\widehat{\log}$ is similar. These declarations are axiomatic so as to make them parametric in the values of $M_{\exp}$, $E_{\exp}$ and such. The definitions of `u_sum` for $\bigoplus$ and `u_lse` for $\widehat{\mathsf{LSE}}$ follows naturally.

The WhyML statement corresponding to Theorem II.5 is given in Figure 3. Preconditions on lines 2–3 express that the size of the array is between 1 and 2^{51}. The precondition on lines 4–6 expresses the bound on the array elements, and the precondition on lines 7–10 expresses Hypothesis (5). the post-condition on lines 11–16 expresses the bound on accuracy given by Theorem II.5. The text of WhyML proof of Theorem II.5 is given by the body of the `lse_accuracy` lemma function, displayed in Figure 4. It more or less follows the paper proof detailed in our report [23]. Notice on lines 3–12 the invocation of Lemma II.3, and on lines 36–42 the invocation of Lemma II.4.

```
1  let ghost s : udouble = u_sum (u_exp_fun a) 0 size in
2  let ghost sum_exps : real = sum (exp_fun a) 0 size in
3  begin
4    (* statement corresponding to Lemma (II.3) *)
5    ensures {
6      abs ((to_real s) - sum_exps) <=
7        sum_exps * (exp_error +
8          eps * from_int (size - 1) * (1.0 + exp_error)) }
9    (* invocation of Lemma (II.3) proved earlier *)
10   u_sum_accuracy_combine_pos
11     exp_error 0.0 (exp_fun a) (u_exp_fun a) 0 size;
12 end;
13 begin
14   ensures { sum_exps > 0.0 }
15   sum_strictly_pos (exp_fun a) 0 size;
16 end;
17 begin (* required domain for calling u_log *)
18   ensures { 0.0 < to_real s <= log_max_value }
19   assert { forall i. 0 <= i < size ->
20     0.0 <= to_real (u_exp (a i)) <=
21       exp max_a * (1.0 + exp_error)
22     by abs (to_real (u_exp (a i)) - exp (to_real (a i)))
23          <= exp (to_real (a i)) * exp_error
24     so to_real (u_exp (a i)) <=
25       exp (to_real (a i)) * (1.0 + exp_error) };
26   (* invocation of Corollary (II.2) *)
27   u_sum_constant_bounds (exp max_a *
28     (1.0 + exp_error)) (u_exp_fun a) size 0 size;
29 end;
30 let ghost r : udouble = u_log s in
31 assert { r = u_lse a size };
32 let ghost err : real = exp_error +
33        eps * from_int (size - 1) * (1.0 + exp_error)
34 in
35 assert { err < 1.0 };
36 begin
37   ensures {
38     abs (log (to_real s) - log sum_exps) <=
39       - log (1.0 - err) }
40   (* invocation of Lemma (II.4) on log *)
41   log_combine_err sum_exps (to_real s) err 0.0;
42 end;
43 assert {
44   (log_error + 1.0) *
45     (abs (log (to_real s) - log sum_exps)) <=
46     - log (1.0 - err) * (log_error + 1.0)
47   by (log_error + 1.0 >= 0.0) };
48 assert {
49   abs (to_real r - lse_exact a size)
50   <= abs (to_real r - log (to_real s)) +
51      abs (log (to_real s) - log sum_exps)
52   <= (log_error + 1.0) *
53        (abs (log (to_real s) - log sum_exps))
54        + log_error * abs (lse_exact a size)
55   <= log_error * abs (lse_exact a size)
56        - log (1.0 - err) * (log_error + 1.0) }
```

Fig. 4. Proof of Theorem II.5 in WhyML.

```
/*@ requires 0 < size <= 1024 && max_a <= exp_max_value;
  @ requires \initialized (&a[0..size-1]);
  @ requires
  @   \forall integer i;
  @     0 <= i < size ==> \abs(a[i]) <= max_a;
  @ // the hypothesis (5) of Theorem II.5
  @ requires
  @   \exp(max_a) * (1.0 + exp_error) * size *
  @            (1.0 + (eps * (size - 1))) <= log_max_value;
  @ // additional requirements to prevent
  @ // overflow on addition
  @ requires max_a <= 701.0;
  @ requires log_max_value <= 0x1p1023;
  @ // result is equal to the WhyML def of LSE on udouble
  @ ensures to_udouble(\result) == u_lse(a, size);
  @ // the accuracy property
  @ ensures \abs(\result - lse_exact(a, size)) <=
  @   log_error * \abs(lse_exact(a,size))
  @   - \log(1 - (exp_error + eps * (size - 1) *
  @            (1 + exp_error))) * (1 + log_error);
  @*/
double log_sum_exp(size_t size) {
 int i;
 double s = 0.0;
 /*@ loop invariant 0 <= i <= size;
   @ loop invariant // to prove the first post-condition
   @    to_udouble(s) == u_sum_of_u_exp(a, 0, i);
   @ // for proving s is the domain of the log
   @ loop invariant (i == 0 ? s == 0.0 : 0.0 < s);
   @ loop invariant
   @   \forall integer j; 0 <= j < i ==>
   @     \abs(to_real(u_exp(to_udouble(a[j])))) <=
   @       \exp(max_a) * (1.0 + exp_error) ;
   @ loop assigns i, s;
   @ loop variant (size - i);
   @*/
 for (i = 0; i < size; i++) {
  /*@ assert 0.0 <= to_real(u_exp(to_udouble(a[i]))) ;
    @ assert to_real(to_udouble(a[i])) <= max_a ;
    @ assert
    @   \exp(to_real(to_udouble(a[i]))) <= \exp(max_a) ;
    @ assert
    @   \abs(to_real(u_exp(to_udouble(a[i])))
    @          - \exp(to_real(to_udouble(a[i]))))
    @      <= \exp(max_a) * exp_error ;
    @ assert
    @   to_real(u_exp(to_udouble(a[i])))
    @     <= \exp(max_a) * (1.0+exp_error) ;
    @ assert   // invocation of Corollary (II.2)
    @   usum_double_bound(u_sum_of_u_exp(a, 0, i),
    @     \exp(max_a) * (1.0 + exp_error), size);
    @*/
  s += exp_approx(a[i]);
 }
 /*@ assert // another invocation of Corollary (II.2)
   @  usum_double_bound(to_udouble(s),
   @    \exp(max_a) * (1.0 + exp_error), size);
   @*/
 return log_approx(s);
}
```

Fig. 5. C code for computing LSE, annotated with ACSL specifications.

To proceed with the proof, we ask Why3 to generate a set of verification conditions (VCs for short). On this lemma function, Why3 generates 30 VCs. All of them except one are proved by the Alt-Ergo SMT solver within a 5 seconds time limit. The only remaining one corresponds to the formula `0 < to_real s` on line 18 of Figure 4, which can be proved instead using the FPA variant of Alt-Ergo [26]. See our report [23] for more technical details on the proofs.

C. Proving a C code implementing LSE

We aim to achieve proofs on concrete C code. For that, we use the environment TIS-kernel, a fork of Frama-C, and its J^3 plug-in for deductive verification, which is a prototype under development. Alternatively, there should be no technical difficulty to achieve the proofs of our C code using the regular Frama-C environment and its Wp plug-in for deductive verification.

Our C code computing the LSE function is given on Figure 5. In a first step, let's ignore the potential floating-point overflow, and focus on proving the accuracy property. To specify the intended behavior and its properties, we build a bridge to WhyML definitions (see our report [23] for technical details), so that for example we can use the WhyML definitions

```
/*@ requires \abs(x) <= exp_max_value;
  @ ensures to_udouble(\result) == u_exp(to_udouble(x));
  @ assigns \nothing;
  @*/
extern double exp_approx(double x);

/*@ requires 0 < x <= log_max_value;
  @ ensures to_udouble(\result) == u_log(to_udouble(x));
  @ assigns \nothing;
  @*/
extern double log_approx(double x);
```

Fig. 6. External C functions for $\widehat{\exp}$ and $\widehat{\log}$, specified in ACSL.

of u_exp and u_log in the ACSL annotations. It provides in particular a function to_udouble that promotes a regular C double to an unbounded double. We declare and specify the auxiliary C functions for computing approximations of $\exp$ and $\log$, as shown on Figure 6.

The first post-condition, on lines 14–15 of Figure 5, thus expresses that the result of the C function is equal to the LSE function defined in WhyML. The second post-condition, on lines 16–20 of Figure 5, expresses the expected bounding property, as stated by Theorem II.5. The second post-condition is going to be proved easily from the first one, and the accuracy result on u_lse already proved in WhyML. The precondition on lines 3–5 is required to allow calling the exponential inside its correct domain. The precondition on lines 6–9 expresses the required hypothesis 5 of Theorem II.5.

The first post-condition on lines 14–15 is an easy consequence of the definition of the LSE function, and the loop invariant given on lines 26–27. The post-condition on lines 16–20 is proved by invoking the proof of the same statement already done in WhyML. Together with the simple loop invariants on lines 25 and 29, all the VCs are proved, in particular the expected post-conditions, except two of them. The first unproved VC is related to line 59 where it is required to show that s fits in the expected domain of the approximated logarithm. The second unproved VC is related to line 53 where it is requires to show the absence of floating-point overflow when performing addition.

To prove the VC on line 59 and thus prove that the sum s on line 59 fits in the expected range of the log, we need to state the additional loop invariants on lines 29 and 30–33 to bound the sum. To prove that these invariants hold, we need again to invoke Corollary II.2. Achieving this proof is a bit involved, requiring all the extra intermediate assertions on lines 38–51.

The last VC remaining to prove is the absence of numerical overflow when computing the addition on line 53. It is indeed expected since the C code operates on true IEEE floating-point numbers and not the unbounded ones. To achieve this, the given pre-conditions are not enough, we need to assume extra bounds on the inputs. So far we assumed the input numbers smaller than $M_{\exp}$, for which no upper bound is assumed so far. Yet, we add the exponentials of these numbers, and summing up to say 1024 of these numbers, we can indeed have an overflow. A tighter bound must be assumed. We assume here, on lines 12–13 of Figure 5, that M_a is smaller than 701 and $M_{\log}$ is smaller than 2^{1023}. With these extra assumptions, and thanks to the already stated and proved loop invariants on lines 29 and 30–33, the VC is proved.

In all, 56 VCs are generated. 49 of them are proved by Alt-Ergo, within a 5 seconds time limit. For the rest, we tried CVC4 and cvc5, which are able to solve 6 VCs, and the last one remaining is proved by the FPA variant of Alt-Ergo.

IV. Related work

As far as we know, the only contribution focusing on rounding errors of LSE-based algorithms is the work of Blanchard *et al.* [10]. They bound the rounding errors of the LSE function and its gradient, namely the softmax function. In this work, the authors assume that the exponential and logarithm functions are implemented with correct rounding, i.e., $E_{\exp} = E_{\log} = \varepsilon$. In contrast, we provide a proof which is parameterized with arbitrary error bounds for the called functions. It means that we can rely on any implementations of these functions without invalidating the bounds. Going back to Theorem II.5, if we take $E_{\exp} = \varepsilon$, we get a bound in which the relative error term is $E_{\log} = \varepsilon$ and the constant term is $-\log(1 - (\varepsilon + (n-1)\varepsilon(1+\varepsilon)))(1+\varepsilon)$, that is, about $-\log(1 - n\varepsilon) + O(u^2)$. Blanchard *et al.*'s relative error term is identical. Their constant term is about $(n+1)\varepsilon + O(u^2)$, which is actually very slightly tighter than ours, but of comparable order of magnitude. Yet, a strength of our accuracy result is that is parametric in the accuracies of $\exp$ and $\log$, instead of assuming ideally precise implementations. Moreover, another main strength compared to this work is the fact that we made formal proofs.

Our work can also be compared to other contributions targeting the end-to-end formal proof of numerical software written in C. For instance, Appel and Bertot [3] have combined the Verifiable Software Toolchain (VST) [2], [4], Flocq [21], [22] and Gappa [30] to formally-verify an example of square root implementation using the Newton method. VST ensures the correctness of the C code, while Flocq and Gappa are used as backend tools to check numerical accuracy facts. Kellison *et al.* [45] propose a Coq formal proofs library, called *LAProof*, for rounding error analysis of basic linear algebra operations, e.g. inner product or matrix-matrix multiplication. As an application example, the authors prove a C program computing a sparse matrix-vector multiplication using the VST [2] approach and the LAProof library.

Boldo *et al.* [13], [15] formalized a numerical integration scheme for a wave partial differential equation in Coq. They not only formally proved a bound on the mathematical errors [15], but also a bound on the rounding errors [13]. These works have been used as a basis for the formal verification of a wave equation resolution C program [16], based on the the Jessie plug-in of Frama-C. The most complex proof obligations related to numerical errors are discharged to Coq.

Becker *et al.* [8] developed a CakeML extension for optimizing floating-point arithmetic in Standard ML. Their approach relies on an end-to-end soundness proof linking a real-number specification of the initial code with the code obtained after optimization. The approach is entirely automated (code and proof generation) and targets the optimization of floating-point kernels, which are essentially blocks of floating-point computations free of control flow instructions. The roundoff errors are obtained and proved by using the prover FloVer (interval-based prover in HOL4).

V. Conclusion and future work

We presented bounds on the accuracy of an implementation of the LSE function. The resulting expressions for the bounds are parametric in the precision of the underlying implementations of $\exp$ and $\log$, and also parameterized by bounds on the size of the argument vectors, and bounds on the values of the vectors components. The given bounds are proved on paper and then in WhyML, using Why3 constructs such as lemma function to provide proofs that follow more or less the paper proofs. We also proved some C implementation by reusing the results proved in WhyML. The proofs are made simpler by using a theory of unbounded floating-point numbers, allowing us to separate the reasoning on accuracy from the reasoning on absence of overflow.

a) Future work.: The bounds exhibited by Blanchard *et al.* [10] show that naive implementations of these functions are relatively well-behaved regarding numerical accuracy, but prone to spurious overflow. The authors then study an alternative implementation to bypass this issue. The principle is to find the maximal value $\alpha = \max(x_i)$ among the components of the input vector and to rewrite the LSE expression as follows:

$$\mathsf{LSE}(x) = \alpha + \log\left(\sum_{1\leq i\leq n} \exp(x_i - \alpha)\right)$$

As for all i, $x_i - \alpha \leq 0$, the exponential takes a reasonable value whatever is the magnitude of the components of x, therefore, the risk of overflow is limited. They also prove that the accuracy of this alternative evaluation is not only as good as with the standard evaluation, but even slightly better. In practice, most applications using the LSE function rely on the shifted version. The proof of the error bounds associated to this alternative evaluation which is presented by Blanchard *et al.* uses rather sophisticated mathematical arguments, *e.g.* Taylor series expansions of the $\log(1+x)$ quantity. Providing a formal proof of this result could be an interesting perspective.

In the presented work, some parts of the proof are performed with a high level of automation. Making formal verification processes automatic and push-button has a strong impact on their industrial applicability. While the studied example is rather intricate and would be difficult to fully automate, there are plenty of applicative source code in which simple combinations of mathematical functions calls appear. For instance, we could try to apply our methodology on benchmarks from the FPBench [27] or COPRIN projects [51]. Most examples from these benchmarks are loop-free which strongly eases the verification process. We could try to handle these examples in a fully automatic way, providing bounds depending on error bounds for the mathematical functions implementations appearing in the source code.

For now, we assume error bounds on the implementations of the mathematical functions $\log$ and $\exp$. Our formally proved bounds apply only when implementations satisfying the assumptions are provided. It is known in the literature that correctly-rounded implementations of these functions can be achieved, *e.g.* in the recent CORE-MATH library [56]. To complement our work, we envision the formal verification of the errors induced by such implementations. In the late 20th century, Harrison [39], [40] formally verified implementations of specific floating-point exponential and trigonometric functions implementations in HOL, but the proofs were *ad hoc*, low-level and far from automatic. More recently, the Gappa tool [30] has been partly used to bound rounding errors of floating-point implementations of functions from the CR-LIBM library [28], [32], [31]. However, proofs are not fully automatic and are devoted to specific implementations. In addition, these works only focus on rounding errors, without taking the mathematical approximation errors into account. Geneau de Lamarlière *et al.* [37] provide a methodology and tooling to ease the formal proofs of low-level floating-point components with a minimal user effort. Their approach relies on a framework for modeling and reasoning on floating-point expressions with some facilities, without neglecting potential exceptional behaviors. For that purpose, they offer tools in the Coq proof assistant to automate the proof of the absence of exceptional behaviors, so that the user can reason on real numbers representation of floating-point expressions. This work is typically applied on mathematical functions implementations, *e.g.* $\exp$ and $\log$. Combining our approach with their methodology would be valuable to complete the toolchain. A longer-term goal could be the development of a formally verified implementation synthesis tool, in the spirit of the Metalibm tools [47], [25]. Metalibm already enables the generation of Gappa scripts certifying the synthesized implementations, but this feature is limited to some pieces of code.

Our work is not limited to the LSE function. We already applied our methodology to others, including the following extension related to computing mutual information:

$$\mathsf{SLSE}(a) = \sum_{0\leq i<n} \log_2\left(\sum_{0\leq j<n} \exp\left(-\frac{(a_i+\rho-a_j)^2}{2}\right)\right)$$

Some bounds on accuracy are already obtained on paper [23]. Yet, we did not yet satisfactorily achieve a formal proof. We identified remaining issues in our proof methodology, that deserve future work. In particular, the formal proofs that we already made on SLSE require a large amount of manual steps, so it is desirable to automate the process. We currently plan to automate the application of so-called “forward propagation lemmas”, which are properties similar to our Lemma II.4 for logarithm, but applied to additions, multiplications, exponential. We believe that automating the application of such lemmas would naturally be reused for proving any code proceeding by composing numerical functions and operations. Another concern is related to the methodogy to deal directly with C code. Our current methodology is far from being usable by non-expert users. This is illustrated for example by the numerous assertions that we had to add in our C code for LSE, on lines 38–51 of Figure 5. There are constructs available in WhyML that would be nice to have at the C level: we think in particular, on one hand, about arbitrary lambda-expressions, and on the other hand the ability to call ghost functions. In fact, ghost functions are in principle present in ACSL [9], but they are limited to ghost C programs, whereas we would need to have ghost *logic* functions that would accept *logic types* are parameters: these include real numbers, unbounded floats, functions (lambda-expressions), etc.

REFERENCES

[1] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010. Tool page at http://www.cl.cam.ac.uk/~lp15/papers/Arith/. doi:10.1007/s10817-009-9149-2.

[2] Andrew Appel. Verified software toolchain. In *European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011. Tool page at https://vst.cs.princeton.edu/. doi:10.5555/1987211.1987212.

[3] Andrew Appel and Yves Bertot. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1):1–16, 2020. URL: https://inria.hal.science/hal-03130704/.

[4] Andrew Appel and Ariel Kellison. Vcfloat2: Floating-point error analysis in coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 14–29, 2024.

[5] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 127–141, Edinburgh, Scotland, July 2010. Springer. URL: http://hal.inria.fr/inria-00534333.

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. http://cvc4.cs.stanford.edu/web/. doi:10.1007/978-3-642-22110-1_14.

[8] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony C. J. Fox. Verified compilation and optimization of floating-point programs in cakeml. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 1:1–1:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ECOOP.2022.1.

[9] Allan Blanchard, Claude Marché, and Virgile Prevosto. *Guide to Software Verification with Frama-C — Core Components, Usages, and Applications*, chapter Formally Expressing what a Program Should Do: the ACSL Language. Springer-Verlag, 2024. URL: https://inria.hal.science/hal-04265707.

[10] Pierre Blanchard, Desmond J Higham, and Nicholas J Higham. Accurately computing the log-sum-exp and softmax functions. *IMA Journal of Numerical Analysis*, 41(4):2311–2330, 2021. doi:10.1093/imanum/draa038.

[11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.gitlabpages.inria.fr/toccata/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[12] Sylvie Boldo. Floats & Ropes: a case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science - ARCoSS*, pages 91–102, Rhodos, Greece, July 2009. Springer. doi:10.1007/978-3-642-02930-1_8.

[13] Sylvie Boldo. Floats and ropes: A case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2009. doi:10.1007/978-3-642-02930-1_8.

[14] Sylvie Boldo. *Deductive formal verification: how to make your floating-point programs behave*. PhD thesis, Université Paris-Sud, 2014.

[15] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal proof of a wave equation resolution scheme: the method error. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2010. URL: http://hal.inria.fr/inria-00450789/en, doi:10.1007/978-3-642-14052-5_12/.

[16] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013. URL: http://hal.inria.fr/hal-00649240/en/, doi:10.1007/s10817-012-9255-4.

[17] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, 2007. URL: https://usr.lmf.cnrs.fr/~jcf/publis/caduceus-floats.pdf, doi:10.1109/ARITH.2007.20.

[18] Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, and Jean-Michel Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023. URL: https://hal.science/hal-04095151, doi:10.1017/S0962492922000101.

[19] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011. URL: http://hal.inria.fr/hal-00777605, doi:10.1007/s11786-011-0099-9.

[20] Sylvie Boldo and Claude Marché. Toccata gallery of verified programs, section "floating-point computations". https://toccata.gitlabpages.inria.fr/toccata/gallery/fp.en.html, 2023.

[21] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. ISTE Press - Elsevier, December 2017. URL: https://hal.inria.fr/hal-01632617.

[22] Sylvie Boldo and Guillaume Melquiond. Some formal tools for computer arithmetic: Flocq and Gappa. In Mioara Joldes and Fabrizio Lamberti, editors, *28th IEEE International Symposium on Computer Arithmetic*, 2021. URL: https://hal.inria.fr/hal-03233227.

[23] Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché, and Raphaël Rieu-Helft. Formally verified bounds on rounding errors in concrete implementations of logarithm-sum-exponential functions. Research Report 9531, Inria, 2023. URL: https://inria.hal.science/hal-04343157.

[24] Sven Brüggemann and Corrado Possieri. On the use of difference of log-sum-exp neural networks to solve data-driven model predictive control tracking problems. *IEEE Control Systems Letters*, 5(4):1267–1272, 2020. doi:10.1109/LCSYS.2020.3032083.

[25] Nicolas Brunie, Christoph Lauter, and Guillaume Revy. Precision adaptation for fast and accurate polynomial evaluation generation. In *30th International Conference on Application-specific Systems, Architectures and Processors*, volume 2160-052X, pages 41–41. IEEE, 2019. doi:10.1109/ASAP.2019.00-32.

[26] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in SMT. In *Computer Aided Verification*, volume 10427 of *Lecture Notes in Computer Science*, pages 419–435, 2017. URL: https://hal.inria.fr/hal-01522770, doi:10.1007/978-3-319-63390-9_22.

[27] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a standard

benchmark format and suite for floating-point analysis. In *Numerical Software Verification*, pages 63–77. Springer, 2017. doi:10.1007/978-3-319-54292-8_6.

[28] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. Cr-libm: a correctly rounded elementary function library. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, volume 5205, pages 458–464. SPIE, 2003.

[29] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. CR-LIBM: a library of correctly rounded elementary functions in double-precision. Research report, LIP, 2006. URL: https://ens-lyon.hal.science/ensl-01529804.

[30] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):1–20, 2010.

[31] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2010.

[32] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using gappa. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1318–1322, 2006.

[33] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[34] Clément Fumex, Claude Marché, and Yannick Moy. Automating the verification of floating-point programs. In Andrei Paskevich and Thomas Wies, editors, *Verified Software: Theories, Tools, and Experiments. Revised Selected Papers Presented at the 9th International Conference VSTTE*, number 10712 in Lecture Notes in Computer Science, Heidelberg, Germany, December 2017. Springer. URL: https://hal.inria.fr/hal-01534533/.

[35] Bolin Gao and Lacra Pavel. On the properties of the softmax function with application in game theory and reinforcement learning. *arXiv preprint arXiv:1704.00805*, 2017.

[36] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. δ-complete decision procedures for satisfiability over the reals. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 286–300. Springer, 2012. doi:978-3-642-31365-3_23.

[37] Paul Geneau de Lamarlière, Guillaume Melquiond, and Florian Faissole. Slimmer formal proofs for mathematical libraries. In Theo Drane and Anastasia Volkova, editors, *Proceedings of the 30th IEEE International Symposium on Computer Arithmetic*, Portland, OR, USA, September 2023.

[38] Cédric Gernigon, Silviu-Ioan Filip, Olivier Sentieys, Clément Coggiola, and Mickaël Bruno. Low-precision floating-point for efficient on-board deep neural network processing, 2023. arXiv:2311.11172.

[39] John Harrison. Floating point verification in hol light: the exponential function. In *International Conference on Algebraic Methodology and Software Technology*, pages 246–260. Springer, 1997.

[40] John Harrison. Formal verification of floating point trigonometric functions. In *International conference on formal methods in computer-aided design*, pages 254–270. Springer, 2000.

[41] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. doi:10.1137/1.9780898718027.

[42] Taocheng Hu and Jinhui Yu. LogSumExp for unlabeled data processing. In *15th International Conference on Software Engineering Research, Management and Applications*, pages 63–69. IEEE, 2017. doi:10.1109/SERA.2017.7965708.

[43] IEEE standard for floating-point arithmetic, 2008. https://dx.doi.org/10.1109/IEEESTD.2008.4610935. doi:10.1109/IEEESTD.2008.4610935.

[44] Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 87:803–819, 2018. URL: https://hal.inria.fr/hal-00934443, doi:10.1090/mcom/3234.

[45] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David S. Bindel. LAProof: A library of formal proofs of accuracy and correctness for linear algebra programs. In *Proceedings of the 30th IEEE International Symposium on Computer Arithhmetic (ARITH)*, September 2023. To appear.

[46] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. doi:10.1007/s00165-014-0326-7.

[47] Olga Kupriianova and Christoph Lauter. Metalibm: A mathematical functions code generator. In *International Congress on Mathematical Software*, volume 8592 of *Lecture Notes in Computer Science*, pages 713–717. Springer, 2014. doi:10.1007/978-3-662-44199-2_106.

[48] Radek Mackowiak, Lynton Ardizzone, Ullrich Kothe, and Carsten Rother. Generative classifiers as a basis for trustworthy image classification. In *Computer Vision and Pattern Recognition*, pages 2971–2981, 2021. doi:10.1109/CVPR46437.2021.00299.

[49] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. doi:10.1017/CBO9781139629294.

[50] Guillaume Melquiond. *Formal Verification for Numerical Computations, and the Other Way Around*. Habilitation à diriger des recherches, Université Paris Sud, April 2019. URL: https://tel.archives-ouvertes.fr/tel-02194683.

[51] Jean-Pierre Merlet. *Parallel Robots*, chapter Structural synthesis and architectures, pages 19–94. Springer, 2006. doi:10.1007/1-4020-4133-0_2.

[52] Taiki Miyagawa and Akinori F Ebihara. The power of log-sum-exp: Sequential density ratio matrix estimation for speed-accuracy optimization. In *International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7792–7804, 2021. URL: https://proceedings.mlr.press/v139/miyagawa21a.html.

[53] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[54] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-point Arithmetic (2nd edition)*. Birkhäuser Basel, July 2018. URL: https://hal.inria.fr/hal-01766584, doi:10.1007/978-3-319-76526-6.

[55] Pierre Roux. Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *Journal of Automated Reasoning*, 57(2):135–156, 2016. doi:10.1007/s10817-015-9339-z.

[56] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH project. In *29th IEEE Symposium on Computer Arithmetic*, pages 26–34, 2022. URL: https://inria.hal.science/hal-03721525, doi:10.1109/ARITH54963.2022.00014.

[57] US Government Accountability Office. Defense patriot missile: Software problem led to system failure at dhahran, saudi arabia. *US Government Accountability Office Reports, rapport no. GAO/IMTEC-92-26*, 1992.

Symbolic Computer Algebra for Multipliers Revisited - It's All About Orders and Phases

Alexander Konrad Christoph Scholl
University of Freiburg, Freiburg, Germany
{konrada, scholl}@informatik.uni-freiburg.de

***Abstract*—Using Symbolic Computer Algebra (SCA) enabled a huge progress in formal verification of arithmetic circuits in recent years. Several different approaches have been proposed showing great success especially for the verification of multipliers. Some of them are based on precomputing and simplifying polynomials for specific circuit structures like converging cones while others take advantage of known or detected hierarchy information to replace and simplify particular sub-circuits of the design. In this paper we propose a new method that avoids the use of such methods and applies only two dynamic approaches: (1) choosing a good substitution order for the backward rewriting process and (2) adjusting the phases of signals occurring in the intermediate polynomials during the verification process. Both methods are simply based on a greedy local search taking the sizes of intermediate polynomials into account. Our experimental results show that this method is very competitive with already existing tools and it improves their robustness, e.g. against optimizations of the verified circuits using logic synthesis.**

I. Introduction

Arithmetic circuits account for an important part in circuit designs, be it general-purpose processors or specialized hardware aimed for computationally intensive applications like cryptography, signal processing or machine learning. The infamous Pentium bug [1] from 1994 raised the communities' awareness about the need of formal methods to verify the correctness of arithmetic circuit designs. Today, the design of arithmetic circuits is not limited to the major processor vendors only, but is also done by various different suppliers of special-purpose embedded hardware who cannot afford to employ large teams of specialized verification engineers being able to provide human-assisted theorem proofs. This results in a growing interest for *fully automatic formal verification* of arithmetic circuits.

Especially the verification of multiplier and divider circuits remained a challenging problem for long time. While BDD-based methods [2], [3] suffer from exponential space complexity, SAT-based methods [4], [5] face exponential run times for larger bit-widths. *BMDs [6]–[8] were presented as a suitable verification data structure for multipliers, but unfortunately *BMDs based verification approaches did not fulfill expectations in practice. Nevertheless, methods based on *Symbolic Computer Algebra (SCA)* have shown great progress for the automatic formal verification of gate-level multipliers and dividers in recent years. They enabled the verification of large and complex arithmetic circuit structures, including finite field multipliers [9], integer multipliers [10]–[25], modular multipliers [26] and divider circuits [27]–[31]. Here the verification task has been reduced to an ideal membership test for the specification polynomial based on so-called backward rewriting, proceeding from the outputs of the circuit in the direction of the inputs. For integer multipliers, SCA-based methods are closely related to verification methods based on word-level decision diagrams like *BMDs, since polynomials can be seen as "flattened" *BMDs [29]. In addition, rewriting based approaches [32], [33] have also shown to be able to verify complex multipliers as well as arithmetic modules with embedded multipliers at the register transfer level.

Most multiplier architectures are basically composed of three stages: (1) Partial Product Generator (PPG), (2) Partial Product Accumulator (PPA) and (3) Final Stage Adder (FSA). Surprisingly, difficulties with exponential polynomial sizes in SCA-based multiplier verification often occurred when using fast adders [34] as the FSAs, and not so often when using complex PPAs. A first hint in this direction was already given by the theoretical analysis for *BMDs in the work of Keim et al. [35]. Most recently three major approaches were published to tackle this problem:

- [18], [19], [21], [24] use reverse engineering and detection of converging cones to precompute polynomials for sub-circuits and simplify those polynomials early on by avoiding so-called *vanishing monomials*.
- [20], [22], [25], [36] use heuristics to detect a parallel prefix adder, replace it by a simple ripple-carry adder and use SAT to prove the soundness of this replacement, changing the multiplier circuit into a structure for which rewriting is much easier.
- [23] uses the adder detection from [20] to determine a parallel prefix adder, but it does not replace the adder by a simpler form. Instead, it introduces *dual variables* and uses a new approach of *carry rewriting* to avoid the occurrence of exponential peak polynomials in the rewriting steps of the parallel prefix adder.

In this paper, we present a new method which consists of two dynamic approaches. First, we advance the idea from [21] of dynamically finding a good substitution order for the backward rewriting process. For this we partition the circuit into blocks and use a hierarchical approach to select a good candidate block for the next substitution step as well as a

This work was supported by the German Research Foundation (DFG) within the project VerA (SCHO 894/5-1).

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_32

good substitution order for the block itself, with the aim of keeping intermediate polynomials as small as possible. Second, we adjust the phases of signals occurring in intermediate polynomials during the backward rewriting with the same goal. Both of our new dynamic approaches work as greedy local search algorithms that only take the current polynomial size into account. As a result, we are not as dependent as other approaches on the detection of specific sub-circuits. This makes our method more robust to circuit optimizations. The simplicity of the approach additionally paves the way for easier certifiability.

The experimental results show that our simple method is very competitive with other existing approaches, being able to verify almost all unoptimized 64-bit multiplier circuits which are at our disposal. However, the main advantage of our method is seen in the verification of optimized benchmarks where we outperform other tools by a large margin.

The paper is structured as follows: In Sect. II we provide background on the basic SCA-method and multiplier circuits. In Sect. III we summarize and discuss existing methods to motivate the need of the novel approach presented in Sect. IV. We evaluate our new approach in Sect. V and conclude with final remarks in Sect. VI.

II. Preliminaries

A. SCA for Verification

For the presentation of SCA we basically follow [29]. SCA-based approaches work with polynomials and reduce the verification task to an ideal membership test using a Gröbner basis representation of the ideal. The ideal membership test is performed using polynomial division. While Gröbner basis theory is very general and, e.g., can be applied to finite field multipliers [9] and truncated multipliers [20] as well, for integer arithmetic it boils down to substitutions of variables for gate outputs by polynomials over the gate inputs (in reverse topological order), if we choose an appropriate "term order" (see [14] or [17], e.g.). Here we restrict ourselves to exactly this view.

For integer arithmetic we consider polynomials over binary variables (from a set $X = \{x_1, \dots, x_n\}$) with integer coefficients from $\mathbb{Z}$, i. e., a polynomial is a sum of terms, a term is a product of a monomial with an integer, and a monomial is a product of variables from X. Polynomials represent *pseudo-Boolean functions* $f : \{0,1\}^n \mapsto \mathbb{Z}$.

As a simple example consider the full adder from Fig. 1. The full adder defines a pseudo-Boolean function $f_{FA} : \{0,1\}^3 \mapsto \mathbb{Z}$ with $f_{FA}(a_0, b_0, c) = a_0 + b_0 + c$. We can compute a polynomial representation for f_{FA} by starting with a weighted sum $2c_0 + s_0$ (called the "output signature" in [13]) of the output variables. Step by step, we replace the variables in polynomials by the so–called "gate polynomials". This replacement is performed in reverse topological order of the circuit, see Fig. 1. We start by replacing c_0 in $2c_0 + s_0$ by its gate polynomial $h_2 + h_3 - h_2h_3$ (which is derived from the Boolean function $c_0 = h_2 \vee h_3$). Finally, we arrive at the polynomial $a_0 + b_0 + c$ (called the "input signature" in [13]) representing the pseudo-Boolean function defined by the circuit. During this procedure (which is called *backward rewriting*) the polynomials are simplified by reducing powers v^k of variables v with $k > 1$ to v (since the variables are binary), by combining terms with identical monomials into one term, and by omitting terms with leading factor 0. We can also consider $a_0 + b_0 + c = 2c_0 + s_0$ as the "specification" of the full adder. The circuit implements a full adder iff backward rewriting, now starting with $2c_0 + s_0 - a_0 - b_0 - c$ instead of $2c_0 + s_0$, reduces the "specification polynomial" to 0 in the end. (This is the notion usually preferred in SCA-based verification.)

The correctness of the method relies on the fact that polynomials (with the above mentioned simplifications resp. normalizations) are canonical representations of pseudo-Boolean functions (up to reordering of the terms). (This is proven in [29], e.g..)

B. Multiplier Circuits

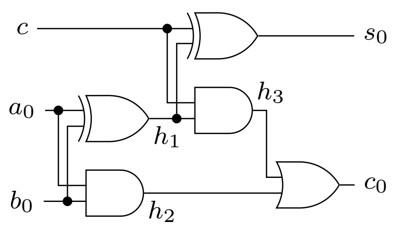
In the following, we briefly summarize textbook knowledge on multipliers. For more details, see [34], e.g.. Most integer multipliers are composed of three stages: The first stage is the *Partial Product Generator (PPG)* which generates partial products from the bits of the two input operands. Examples are *Simple PPGs*, which just compute the logical AND of all bits of the first input and all bits of the second input, or PPGs with *Booth Encoding* which reduce the number of generated partial products using Booth's Algorithm [37]. The second stage is the *Partial Product Accumulator (PPA)* which sums up all the partial products until they are reduced to two numbers only. Well-known accumulation structures are array accumulation, Wallace trees [38] or Dadda trees [39]. The third stage consists of the *Final Stage Adder (FSA)* which converts the resulting two numbers from the PPA stage into the final binary representation of the output product. Any two operand adder networks can be used here, ranging from simple examples such as the well-known ripple-carry adder to more complex structures such as various implementations of parallel prefix adders. Such implementations include the Kogge-Stone adder [40], the Brent-Kung adder [41] and the Ladner-Fischer adder [42], to name just a few.

C. Specification Polynomial for Unsigned Multipliers

In this paper, we focus on unsigned gate-level integer multipliers with input bits $a_0, \dots, a_{n-1}, \ b_0, \dots, b_{n-1}$ of multiplier and multiplicand and output bits $p_0, \dots, p_{2n-1}$ of the product. The corresponding specification polynomial, which is the starting point of the backward rewriting process, is

$$P_{spec}(p_0, \dots, p_{2n-1}, a_0, \dots, a_{n-1}, b_0, \dots, b_{n-1}) = \sum_{i=0}^{2n-1} 2^i p_i - (\sum_{j=0}^{n-1} 2^j a_j) \cdot (\sum_{k=0}^{n-1} 2^k b_k). \quad (1)$$

As explained in Sec.II-A the multiplier circuit is correct iff backward rewriting reduces P_{spec} to 0.



$$
\begin{array}{rcl}
c_0 & = & h_2 + h_3 - h_2h_3 \\
s_0 & = & c + h_1 - 2ch_1 \\
h_3 & = & ch_1 \\
h_2 & = & a_0b_0 \\
h_1 & = & a_0 + b_0 - 2a_0b_0 \\
\hline
 & & 2c_0 + s_0 \\
 & \xrightarrow{c_0} & 2h_2 + 2h_3 - 2h_2h_3 + s_0 \\
 & \xrightarrow{h_3} & 2h_2 + 2ch_1 - 2ch_1h_2 + s_0 \\
 & \xrightarrow{s_0} & 2h_2 - 2ch_1h_2 + c + h_1 \\
 & \xrightarrow{h_2} & 2a_0b_0 - 2a_0b_0ch_1 + c + h_1 \\
 & \xrightarrow{h_1} & a_0 + b_0 + c
\end{array}
$$

Fig. 1: Full adder circuit with series of substitutions.

In the SCA-based verification for integer arithmetic we use terms with coefficients from $\mathbb{Z}$ in the polynomials. However, for n-bit integer multipliers the polynomial computations can be performed in $\mathbb{Z}_{2^{2n}}$ instead of $\mathbb{Z}$ which is desirable, since it improves efficiency by reducing the maximal coefficient size. In this case the specification polynomial can be defined as

$$
P_{spec,mod}(p_0,\ldots,p_{2n-1},a_0,\ldots,a_{n-1},b_0,\ldots,b_{n-1}) = \left(\sum_{i=0}^{2n-1} 2^i p_i - (\sum_{j=0}^{n-1} 2^j a_j) \cdot (\sum_{k=0}^{n-1} 2^k b_k) \right) \mod 2^{2n}. \quad (2)
$$

In the following paragraph, we give the sketch of a proof that a circuit fulfills the specification from Eqn. (1) iff it fulfills the specification from Eqn. (2) (a similar proof is given in [20]): Consider the polynomial P_{spec} from Eqn. (1). For all possible assignments to $p_i, a_j, b_k \in \{0,1\}$ it holds that P_{spec} evaluates to a value in $\{-(2^n-1)^2,\ldots,2^{2n}-1\}$ (which is easy to see since the upper bound is reached for $p_0,\ldots,p_{2n-1}=1$ and $a_0,\ldots,a_{n-1},b_0,\ldots,b_{n-1}=0$, and the lower bound is reached for the opposite case). By replacing variables by their gate polynomials (without modulo 2^{2n}) we obtain functions depending on a different set of variables, but their range is still a subset of the range of P_{spec}. In the end (after all substitutions) we get a function whose range is still a subset of $\{-(2^n-1)^2,\ldots,2^{2n}-1\}$ (regardless of whether the circuit is correct or not). If we now apply a modulo 2^{2n} operation on those values, they might change, but still 0 will be mapped to 0 and values different from 0 will be mapped to values different from 0, since the absolute values in $\{-(2^n-1)^2,\ldots,2^{2n}-1\}$ are all smaller than 2^{2n}. Therefore it holds: All assignments consistent with some circuit C evaluate P_{spec} to 0 iff all assignments consistent with C evaluate $P_{spec,mod}$ to 0. This means that we can perform all polynomial computations in $\mathbb{Z}_{2^{2n}}$ instead of $\mathbb{Z}$ during backward rewriting.

III. Review of Existing Approaches

A large number of excellent and non-trivial methods has been presented in the literature to enable SCA-based multiplier verification and increase its efficiency. Most of those methods are tailored towards certain structural properties of existing multiplier circuits. Here we will review and analyze existing approaches from the literature. Some of those techniques mentioned here have been described in the literature based on computations with Gröbner bases (like the elimination of certain polynomials from Gröbner bases). Here we prefer a description based on backward rewriting, as already mentioned in Sect. II-A. Note however that the difference is only in the representation, not in the actual contents.

The first approach is the *detection of so-called "atomic blocks"* in the multiplier design [19]. Atomic blocks may be XOR gates, half adders (HAs), full adders (FAs), or Compressors (CMs). In [19], [24] and [17] the information about atomic blocks is used to enable a *hierarchical polynomial computation*: Polynomials for atomic blocks are computed first and during the backward rewriting the local polynomials for atomic blocks are used for replacements in the global specification polynomial. If atomic blocks have several outputs, then either polynomials for the outputs are computed separately and are handled as mentioned above, or the "word-level" polynomial for the atomic block (like $2c_0 + s_0 = a_0 + b_0 + c$ for a FA) is used for rewriting the global specification polynomial, provided that it has an appropriate form (in the case of the FA mentioned above: if the only terms containing c_0 and s_0 in the specification polynomial are $2^{k+1}c_0$ and $2^k s_0$).

Note that in our work we use atomic block detection as well, but we always perform replacements by backward rewriting on the gate level, not on the block level. Atomic blocks are only used to guide the order in which we replace the gates.

Sayed-Ahmed et al. [14] observed that polynomials can be simplified based on the observation that the sum outputs s and the carry outputs c of HAs can never be 1 at the same time. Therefore, terms containing both c and s (called "*vanishing monomials*") can be removed from the polynomial immediately. The authors of [14], [18] argue that vanishing monomials are the main cause of polynomial size explosions during backward rewriting of multipliers. This technique is used in other works like [15] as well. In [18] the observation was used in the context of a hierarchical polynomial computation. [18] computes so-called *Convergent Gate Cones* (CGCs) which are logic cones where paths from outputs of the same half adder (HA) converge in a common node. The polynomials of CGCs are precomputed, vanishing monomials are removed during this computation, and the "cleaned" polynomials for CGCs are used during the final backward rewriting. If large CGCs occur, the advantage of this method is not clear at first sight, since the vanishing monomials can only be removed at the end of backward rewriting of a CGC. However, in fast final stage adders (FSAs) like carry-lookahead or parallel prefix adders there are usually overlapping CGCs of different sizes. It is essential that the method from [18] starts with the smaller CGCs, cleans them, and uses the resulting polynomials when polynomials for larger, overlapping CGCs are computed. In [26] the approach has been generalized to arbitrary pairs of contradicting signals.

The idea of a hierarchical polynomial computation has also been applied in [21], [24] to *fanout-free cones* of gates which are not included in atomic blocks and CGCs. This technique has been called "*Common Term Rewriting*" in [14].

The special treatment of CGCs was motivated by difficulties of SCA-based methods when the FSA is a parallel prefix adder. Kaufmann et al. [20], [23] propose other techniques to tackle the same problem. They substitute a final parallel prefix adder by a simple ripple-carry adder and then verify the resulting simplified multiplier. In [20] the equivalence of the substituted parallel prefix adder with a ripple-carry adder is proven by SAT solving. [23] introduces the concepts of *dual variables* and *tail substitution* and uses them during "*carry rewriting*" inside the detected FSA. For both [20] and [23] certain circuit structures inside the FSA as well as the FSA bounds have to be structurally detected. Those methods work well and are fast for clean multipliers (as the circuit rewriting from [32], [33] as well), but they become problematic, if logic synthesis steps at the gate level have destroyed the clear structure. This observation is clearly confirmed by our experiments in Sect. V.

Finally, the order of traversing the circuit during backward rewriting is of utmost importance. Reverse topological orderings are usually not unique, but leave a lot of degrees of freedom. Different orderings may lead to totally different sizes of the intermediate polynomials during backward rewriting. Some methods try to find good static orders for the traversal (and sometimes need structural information about the circuit to find them). [13] performs a *"row-wise" traversal* of the multiplier (which resembles a breadth-first search in the circuit, augmented with information on hierarchy bounds), [15] performs a *"column-wise" traversal* (which resembles a depth-first search). In [15] a decomposition of the multiplier specification polynomial into column-wise "sub-specifications" is used in addition (needing an additional multiplier-specific reasoning for correctness). In contrast, in [21] a dynamic substitution order was proposed. Here the order in which backward rewriting processes the circuit is not determined beforehand, but it is adjusted dynamically based on the sizes of intermediate polynomials. It is important to note that the dynamic substitution order is used in the context of hierarchical polynomial computation on the level of "components" (atomic blocks, CGCs, fanout-free cones) here. The substitution orders to compute polynomials for the components are still chosen statically.

Most of the approaches mentioned above work well, when they are applied to clean multipliers at the gate level where logic synthesis has not been applied, but they become weaker, when they are applied to multipliers which are optimized by logic synthesis. This does not only hold for methods which rely on the detection of FSA boundaries and FSA structures as mentioned above [20], [23], but also for other methods: The hierarchical polynomial computation becomes weaker, if the boundaries of atomic blocks are destroyed by logic synthesis; the removal of vanishing monomials based on CGCs gets into trouble, if HAs at the origin of CGCs cannot be detected anymore due to logic restructuring; the computation of static replacement orders may suffer as well, if the circuit structures are destroyed on which the order computation relies (like XOR-skeletons for the computation of column-wise slices in [25]).

For this reason, we investigate in this paper whether it is possible to increase the robustness of SCA-based verification of multipliers by avoiding complex approaches which are vulnerable against changes to the clean multiplier structure. Our goal is to simplify the approach and at the same time to increase its robustness. We restrict ourselves to a flat and non-hierarchical polynomial computation based on backward rewriting with gate polynomials, but we invest considerable effort into the computation of good substitution orders. Moreover, we deeply integrate the technique of phase optimization into our order optimization to make the approach more robust against the selection of unfavourable orders.

IV. Backward Rewriting with Phase and Order Optimization

In this section we present our SCA-based verification method which is based on two simple ingredients: The optimization of "phases" of signals and the optimization of the order of gate replacements during backward rewriting. Both methods are integrated into an overall phase and order optimization, but we begin with the description of phase optimization.

A. Phase Optimization

The idea of phase optimization is to adjust the phases of occurring signals during backward rewriting to keep intermediate polynomial sizes small and to make backward rewriting more robust, e.g., against different orderings. The approach is based on the observation that replacing variables by their negation in intermediate polynomials may reduce the sizes of those polynomials.

Example 1. *Let us consider the function $f(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$ which computes the disjunction of 3 inputs. It is easy to see that the polynomial for f is $p = x_1 + x_2 + x_3 - x_1x_2 - x_1x_3 - x_2x_3 + x_1x_2x_3$ with 7 terms. Now we change the phase of one variable, let's say x_1, i.e., we introduce a "new input variable" called $\overline{x_1}$ representing the negation of x_1 and replace x_1 in p with $1 - \overline{x_1}$. This results in $p' = 1 - \overline{x_1} + \overline{x_1}x_2 + \overline{x_1}x_3 - \overline{x_1}x_2x_3$ with 5 terms. Changing the phase of x_2 (i.e. replacing x_2 with $1 - \overline{x_2}$) results in $p'' =$*

Algorithm 1 Phase Optimization.

```
Input: Polynomial $P$, Set of candidate signals $S$
Output: Polynomial $P$ with optimized phases
1: for each $s \in S$ do
2:     $old\_size \leftarrow$ size($P$);
3:     $P \leftarrow$ Flip-Phase($s, P$);                    ▷ flip signal $s$ in $P$
4:     $new\_size \leftarrow$ size($P$);
5:     if $old\_size < new\_size$ then $P \leftarrow$ Flip-Phase($s, P$)  ▷ flip $s$ back
6: return $P$;
```

$1 - \overline{x_1}\,\overline{x_2} + \overline{x_1}\,\overline{x_2}x_3$ and changing the phase of x_3 finally results in $p''' = 1 - \overline{x_1}\,\overline{x_2}\,\overline{x_3}$ with 2 terms. The example shows that phase optimization is able to reduce the sizes of polynomials. If $f(x_1, x_2, x_3) = x_1 \vee x_2 \vee x_3$ is part of a larger circuit and x_1 has to be replaced with the polynomial of another gate g, then the phase flipping of x_1 has of course to be reverted before replacing x_1 with the polynomial of g. Anyway, our hope is that phase optimization is able to reduce the size of intermediate polynomials. If g is constant 1, e.g., then reverting the phase flipping of x_1 results in $1 - \overline{x_2}\,\overline{x_3} + x_1\overline{x_2}\,\overline{x_3}$ and after replacing x_1 with constant 1 we arrive at $p'''' = 1$. The experimental results in Sect. V show that our hope for the benefits of phase optimization is well founded.

We perform phase optimization by a simple greedy algorithm, see Alg. 1. The algorithm is a general function of the polynomial package and, therefore, does not need any circuit information, but only uses the polynomial which should be optimized and a set of candidate signals which are objective to the optimization. For every candidate signal s the following is done: Saving the current size of the polynomial P (which is just the number of terms in P, line 2), then flipping the phase of the signal s in P (line 3). If the now achieved polynomial size is larger than before, the phase change was not beneficial and s is flipped again to restore the previous polynomial (line 5). Only if the polynomial size could have been reduced by the phase change, the flipped phase is kept in the polynomial and the algorithm continues with the next candidate signal. In the end, a smaller polynomial with optimized phases is found or (in case no phase changes led to a smaller size) the original polynomial remains. Therefore our phase optimization never increases the polynomial size. In our implementation, we perform phase optimization after each rewriting step. To save computation time, we restrict the search space of the phase optimization by choosing the set S of candidate signals in Alg. 1 as the set of variables which were newly introduced into the polynomial in the last rewriting step.

The correctness of flipping phases of signals during backward rewriting can be seen easily: Consider some arbitrary gate g of a circuit computing the signal x_i. Phase flipping of x_i can be simulated by inserting two consecutive inverters immediately at the output of g (which apparently does not change the function of the circuit). The signal after the first inverter is called $\overline{x_i}$, the signals before the first inverter and after the second inverter are called x_i. Phase flipping for x_i corresponds to backward rewriting of the second inverter (introducing $\overline{x_i}$ into the polynomial). Reverting phase flipping before replacement of g corresponds to backward rewriting of the first inverter (introducing x_i into the polynomial again).

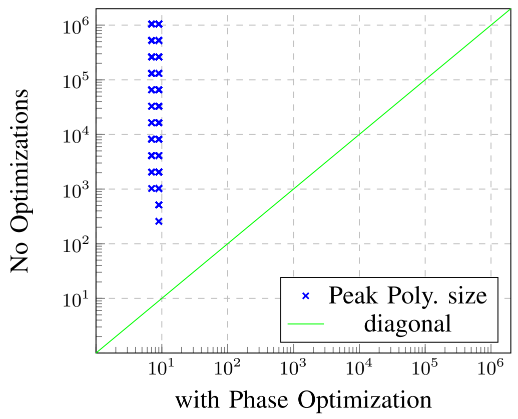


Fig. 2: Peak polynomial sizes for different orderings.

Before showing how we integrate phase optimization into our dynamic order optimization approach, we present an example which shows that phase optimization makes backward rewriting much more robust against changes of replacement orders. The example is a toy example chosen for illustration, but the demonstrated effect is similar to the effects occurring during backward rewriting of parallel prefix adders which contain large OR trees in their implementation. Large OR trees may be a problem for SCA-based methods due to their exponential polynomial representation, see also [20] and [23].

Example 2. *Consider a circuit with 32 input signals $i_0, \ldots, i_{31}$. The 32 input signals are connected to 32 inverters and the outputs of inverters are the inputs of a balanced OR tree. It is clear that this circuit basically implements a NAND function with 32 inputs. Thus, backward rewriting starting with the polynomial $p_o = o$ for the output variable o leads to the final polynomial $p = 1 - i_0 \cdot \ldots \cdot i_{31}$. The final polynomial has size 2 (number of terms).*

Backward rewriting proceeds in reverse topological order and of course there is a huge number of different possible reverse topological orders to traverse the circuit. In our experiment, we randomly choose one of the possible reverse topological orders and perform backward rewriting with and without phase optimization. We repeat the experiment 10.000 times with different random choices. Fig. 2 shows a scatter plot with each data point representing one possible traversal order. The y-axis indicates the peak polynomial size during backward rewriting using this order without *phase optimization and the x-axis* with *phase optimization (note that the axes are scaled logarithmically). Fig. 2 shows that with phase optimization the peak polynomial size is always between 7 and 9. Without phase optimization, the peak polynomial sizes vary considerably and in 698 out of 10.000 cases our chosen upper bound of 1.000.000 terms is exceeded. The large intermediate sizes of the polynomials in the version without phase optimization can be easily explained by the fact that the intermediate polynomials often represent disjunctions of a large number of inputs* *which have exponential representations as polynomials [23]. Apparently, phase optimization keeps intermediate polynomial sizes small and makes the backward rewriting much more robust.*

Our phase optimization is related to the method from [23] using dual variables. In contrast to [23] we do not introduce dual phases x_i and $\overline{x_i}$ of variables, but we only enable flipping the phases of variables, i.e., we flip *all* occurrences of a signal in the polynomial. This makes the approach much simpler and does not need additional handling and simplification steps in the polynomial (like the merging algorithm in [23] which checks if two terms can be merged into one because they only differ in one dual variable).

Interestingly, as also observed in [43], phase optimization is strongly related to replacing the so-called *positive Davio decomposition* by *negative Davio decomposition* in K*BMDs [44]. In contrast to K*BMDs, *BMDs [6] only allow positive Davio decomposition. The positive Davio decomposition wrt. variable x_1 decomposes a function $f : \{0,1\}^n \mapsto \mathbb{Z}$ according to $f = f_{x_1=0} + x_1 \cdot (f_{x_1=1} - f_{x_1=0})$. $f_{x_1=0}$ and $f_{x_1=1}$ are the functions resulting from f by replacing the variable x_1 by 0 and 1, respectively. The correctness of the decomposition can be easily shown by case distinction wrt. $x_1 = 0$, $x_1 = 1$. *BMDs are graphs which basically represent the "sub-functions" $f_{x_1=0}$ and $f_{x_1=1} - f_{x_1=0}$ by nodes. The representation of Boolean polynomials without phase flipping corresponds to positive Davio decomposition: If p is the polynomial for f, then the polynomial for $f_{x_1=0}$ contains all terms which do not include x_1 (in Example 1 $x_2 + x_3 - x_2x_3$) and $x_1 \cdot (f_{x_1=1} - f_{x_1=0})$ contains all terms which do include x_1 (in Example 1 $x_1 - x_1x_2 - x_1x_3 + x_1x_2x_3 = x_1 \cdot (1 - x_2 - x_3 + x_2x_3)$). Thus, *BMDs can be seen as a *factored form* of Boolean polynomials. The *negative* Davio decomposition wrt. x_1 decomposes f according to $f = f_{x_1=1} + (1-x_1)\cdot(f_{x_1=0} - f_{x_1=1})$. By replacing $1-x_1$ with $\overline{x_1}$ it is easy to see that negative Davio decomposition corresponds to Boolean polynomials with phase flipping for x_1. In Example 1 $f_{x_1=1} = 1$, $f_{x_1=0} - f_{x_1=1} = -1 + x_2 + x_3 - x_2x_3$, and $f_{x_1=1} + \overline{x_1} \cdot (f_{x_1=0} - f_{x_1=1}) = 1 + \overline{x_1} \cdot (-1 + x_2 + x_3 - x_2x_3) = p'$.

Whereas [43] performs phase optimization as well, its optimization approach is pretty complex: It translates polynomials into K*BMDs, then uses a K*BMD minimization method [45] that changes the decomposition types (positive or negative Davio) of the variables, and finally it translates the K*BMDs back into polynomials – in the hope that size reductions in K*BMDs translate into size reductions of polynomials. As already mentioned above, our implementation of phase optimization is much simpler: It greedily optimizes the phases of variables and additionally restricts the phase optimization to variables newly introduced into the polynomial in the last rewriting step.

B. Backward Rewriting with Dynamic Order Optimization

Now we introduce our new method of backward rewriting with Dynamic Order Optimization. It can be seen as an improvement of the algorithm from [21] which was the first work to introduce dynamic rewriting in a SCA context.

Algorithm 2 Rewriting with Dynamic Order Optimization.

```
Input: Specification polynomial $SP^{init}$; Circuit $CUV$
Output: TRUE iff specification holds
 1: $SP_i \leftarrow SP^{init}$;
 2: $A \leftarrow$ Detect-Atomic-Blocks($CUV$);
 3: $B \leftarrow$ Compute-EABs($A, CUV$);
 4: for each $b \in B$ do
 5:     $Penalty[b] \leftarrow 1$;
 6: $C \leftarrow$ Get-First-Candidate-EABs($B$);
 7: while $C$ is not empty do
 8:     for each $c \in C$ do
 9:         $Score[c] \leftarrow$ Count-Occurrence($c$, $SP_i$) $\cdot$ $Penalty[c]$;
10:     $sortedC \leftarrow$ Sort-Candidates-Ascending($C$, $Score$);
11:     $SP_{old} \leftarrow SP_i$;   $chosen \leftarrow 0$;
12:     $j \leftarrow 0$;   $upB\_fac \leftarrow 1$;   $best\_j \leftarrow -1$;
13:     while $chosen = 0$ do
14:         $SP_i \leftarrow SP_{old}$;
15:         $(success, SP_i) \leftarrow$ Rewrite($SP_i$, $sortedC[j]$, $upB\_fac$);
16:         if success = TRUE then
17:             $threshold \leftarrow 0.01 \times$ Get-Node-Count($sortedC[j]$);
18:             $growth \leftarrow$ (size($SP_i$) $-$ size($SP_{old}$))/size($SP_{old}$);
19:             if $growth < threshold$ then
20:                 $chosen \leftarrow sortedC[j]$;
21:             else
22:                 $best\_j \leftarrow$ Save-Best-Candidate-So-Far($SP_i, j$);
23:                 $Penalty[sortedC[j]] \leftarrow Penalty[sortedC[j]] \times 2$;
24:         else
25:             $Penalty[sortedC[j]] \leftarrow Penalty[sortedC[j]] \times 2$;
26:         $j \leftarrow j + 1$;
27:         if $chosen = 0$ and $j =$ size(sortedC) then
28:             if $best\_j \geq 0$ then
29:                 $chosen \leftarrow sortedC[best\_j]$;
30:                 $(success, SP_i) \leftarrow$ Rewrite($SP_i$, $chosen$, $upB\_fac$);
31:             else
32:                 $j \leftarrow 0$;   $upB\_fac \leftarrow upB\_fac \times 2$;
33:     $C \leftarrow$ Update-Candidates($B$, $C$, $chosen$);
34: if size($SP_i$) $= 0$ then return TRUE else return FALSE;
```

The observation is that during backward rewriting, usually, there are several candidates (be it on the lower level of individual nodes or on a higher "component" level like atomic blocks or fan-out-free cones) to choose from for the next substitution step. The goal is always to find a substitution order which keeps intermediate polynomials small. While it is easier to find "good" *static orderings* (which are computed before the rewriting process started) for clean multiplier circuits, this task gets hard for optimized circuits where the clean boundaries between different functional blocks might vanish, e.g. due to applied logic synthesis. The idea of Dynamic Order Optimization is to take the current polynomial into account to choose good candidates for the next substitution step. We combine this idea with our new phase optimization approach to achieve a stronger dynamic ordering.

We start with a rough overview of our Dynamic Order Optimization: Basically, we use *two* dynamic procedures on different hierarchy levels to obtain a good dynamic order for backward rewriting. On the higher "component level" (see Alg. 2) we use dynamic ordering on the level of components, choosing a good candidate component in every step. In our case the components are so-called Extended Atomic Blocks

(EABs) [31]. On the lower level (see Alg. 3) we use dynamic ordering as well, but now on the level of individual circuit nodes inside the components. This is an important difference to the approach from [21] which uses a dynamic ordering approach on the component level, but a static approach for precomputing polynomials of those components (which are used later on in a global rewriting procedure).

Now we come to a more detailed description of our method: Our algorithm for backward rewriting with Dynamic Order Optimization is depicted in Algs. 2 and 3. We start in Alg. 2 with the specification polynomial SP^{init} and the circuit under verification CUV in AIG format. In the beginning we detect atomic blocks (XORs, HAs, FAs, as well as single sum and carry outputs of FAs) (see line 2). Next, we combine these atomic blocks and the remaining gates of the circuit into Extended Atomic Blocks (EABs) [31] (line 3). The idea of EABs is to combine atomic blocks and remaining gates into fanout-free cones to partition the circuit into functionally and structurally related subcircuits. While [31] uses EABs to compute don't cares on their inputs to optimize polynomials during backward rewriting and [21] uses a similar concept called "components" with the purpose of locally precomputing polynomials for those components before using them in a global rewriting procedure, we use EABs only for the purpose of helping to find good substitution orders.

For every EAB we initialize a penalty factor of 1 (line 5). Next the initial list of candidate EABs is computed (line 6). To do substitutions in reverse topological order, this list contains all EABs which only have fan-outs into primary outputs but not into any other EABs. Our method can be seen as a two-leveled, hierarchical ordering approach. The first, upper level, working on the level of EABs, is described by the while loop from lines 7 to 33. In each round one candidate EAB for substitution is picked in a dynamic fashion and the candidate set is adjusted, until all EABs and therefore the complete circuit has been substituted. In detail this works as follows: A score is assigned to each candidate which is computed as the number of occurrences of the candidate's output signals in SP_i multiplied by the individual penalty factor of the candidate (lines 8 and 9). Afterwards, the candidates are sorted by their scores in ascending order (line 10). The intuition is to prefer candidates with small numbers of output signal occurrences in the polynomial, since from a worst-case perspective many occurrences of EAB output signals in the current polynomial lead to a higher risk of a large polynomial after substitution of this EAB, see also [21], Example 6. Here this idea is adjusted by an additional penalty factor which will be explained later.

In the inner while loop from lines 13 to 32 the actual selection of the candidate EAB happens by iterating through the sorted candidates until a suitable one has been chosen. First, the old polynomial is restored. Next, the actual backward rewriting steps are performed for the current candidate (line 15). The details of how an EAB is rewritten are shown in Alg. 3 and will be explained later. At this point it is only important that rewriting may succeed or fail due to size limitations, which will be indicated by the return value $success$. If it succeeded, a polynomial is returned where all nodes from the current candidate have been rewritten. Otherwise the rewriting has been aborted. In the successful case a threshold (line 17) defines how much growth of SP_i is acceptable for the current candidate. This threshold increases with the number of AIG nodes in the candidate EAB to avoid a bias towards the selection of small EABs (whose replacement cannot increase the polynomial too much). If the growth of SP_i stays below the threshold the candidate is accepted (line 20). Otherwise it is checked if the current candidate produced the smallest polynomial size so far and is therefore saved as best candidate until now (line 22). The penalty factor of the current candidate EAB is increased, if it either has not been accepted (line 23) or the rewriting of the candidate even failed due to the size limit (line 25). The idea of a penalty factor is to avoid that the same candidates get checked unsuccessfully over and over again, because their occurrence count is small, but their substitution is very costly all the same.

In case all candidates have been checked and none of them stayed below the threshold (line 27), the best found candidate is rewritten again. This way we do not need to repeat the complete iteration as long as one candidate could have been rewritten successfully. Only in the case that none of the candidates could have been rewritten without aborting, the upper bound factor upB_fac used for the internal rewriting of Alg. 3 is increased (line 32) and the iteration starts with the same set of candidates again. By this increasing it is guaranteed that eventually some candidate can be rewritten successfully (assuming unlimited resources). After a candidate was chosen, the set of candidate blocks is adjusted (line 33): The chosen candidate is removed and new candidate EABs, which are fan-ins of the just chosen one might get included, if all of their fan-out EABs have already been substituted. At the end the algorithm returns TRUE if SP_i has been reduced to 0, meaning the specification is fulfilled by the circuit.

Next, we explain the actual rewriting process for an EAB which can be seen as the second, lower level of our hierarchical ordering approach working on the level of individual nodes inside of EABs. The algorithm is shown in Alg. 3. As inputs it takes the current polynomial SP_i, an EAB E and some factor upB_fac. An upper bound for intermediate polynomial sizes is computed (line 2) as follows: If $10 \times \text{size}(SP_i) < 100,000$ the upper bound is set to $10 \times \text{size}(SP_i)$, otherwise it is set to $\text{size}(SP_i) + 100,000$. Additionally the upper bound is multiplied by the upB_fac factor afterwards.

Before computing an order dynamically, the algorithm first tries rewriting with two predefined orders based on breadth-first-search and depth-first-search (line 3), with applying phase optimization after every node rewriting. It is history dependent (based on successful rewriting for the current EAB in the past) which order is tested first, and if the first order fails (due to exceeding the upper bound limit) the second is tried. In case one of the orders was successful, TRUE is returned together with the rewritten polynomial SP_i.

Only if none of the two predefined orders was successful,

Algorithm 3 Dynamic Rewriting of an EAB.

```
Input: Polynomial $SP_i$, EAB $E$, Factor $upB\_fac$
Output: ((TRUE if successful, FALSE otherwise), Polynomial $SP_i$)
 1: $SP_{start} \leftarrow SP_i$;
 2: $upperBound$ = Compute-Upper-Bound(size($SP_i$), $upB\_fac$);
 3: $(success, SP_i) \leftarrow$ Try-Rewriting-With-BFS-DFS-Orders($SP_i$, $E$);
 4: if $success = FALSE$ then
 5:     $SP_i \leftarrow SP_{start}$;
 6:     $success \leftarrow TRUE$;
 7:     $N \leftarrow$ Get-First-Candidate-Nodes(nodes($E$));
 8:     while $N$ is not empty and $success = TRUE$ do
 9:         for each $n \in N$ do
10:             $Score[n] \leftarrow$ Count-Occurrence($n$, $SP_i$);
11:         $sortedN \leftarrow$ Sort-Candidates-Ascending($N$, $Score$);
12:         $SP_{old} \leftarrow SP_i$;   $chosen \leftarrow 0$;   $j \leftarrow 0$;
13:         while $chosen = 0$ do
14:             $SP_i \leftarrow SP_{old}$;
15:             $SP_i \leftarrow$ Rewrite($SP_i$, $sortedN[j]$);
16:             $SP_i \leftarrow$ Phase-Opt($SP_i$, Get-Inputs($sortedN[j]$));
17:             $threshold \leftarrow 0.1$;
18:             $growth \leftarrow$ (size($SP_i$) $-$ size($SP_{old}$))/size($SP_{old}$);
19:             if $growth < threshold$ then
20:                 $chosen \leftarrow sortedN[j]$;
21:             else
22:                 $best\_j \leftarrow$ Save-Best-Candidate-So-Far($SP_i, j$);
23:             $j \leftarrow j + 1$;
24:             if $chosen = 0$ and $j$ = size(sortedN) then
25:                 $chosen \leftarrow sortedN[best\_j]$;
26:                 $SP_i \leftarrow$ Rewrite($SP_i$, $chosen$);
27:                 $SP_i \leftarrow$ Phase-Opt($SP_i$, Get-Inputs($chosen$));
28:         $N \leftarrow$ Update-Candidates(nodes($E$), $N$, $chosen$);
29:         if size($SP_i$) $> upperBound$ then
30:             $success \leftarrow false$;
31:             $SP_i \leftarrow SP_{start}$;
32: return $(success, SP_i)$
```

dynamic ordering is started. The dynamic ordering on the node level works very similar to the dynamic ordering on EAB level of Alg. 2, thus we will keep the description short here. Again candidates, which are individual circuit nodes here, are sorted based on a scoring and we look for a candidate which can keep the relative polynomial growth below a threshold (which is a fixed value of 0.1 here) or, if such candidate does not exist, the best found so far is picked. This is repeated until all nodes have been rewritten. After each (tentative or final) node rewriting we immediately apply phase optimization to the newly introduced signals (lines 16, 27). There is one special case for nodes of the EAB being part of an atomic block (XORs, HAs, FAs, single FA outputs) that is not explicitly represented in Alg. 3: They are only rewritten if *all* output nodes of the atomic block are in the candidate set N and whenever one of the output nodes of the atomic block should be rewritten, then all the nodes in the atomic block are rewritten in a fixed precomputed order. The major difference to Alg. 2 is that after every chosen candidate node it is checked whether the current polynomial size exceeds the upper bound (line 29). If this is the case, the dynamic rewriting of this EAB E is aborted and returns FALSE together with the starting polynomial SP_{start}. Therefore, the rewriting of an EAB is not always completed. This is used in Alg. 2 to avoid investing too much space and time in the rewriting of one specific candidate EAB, while there may be other suitable EABs to choose from.

C. *Simple Certifiability*

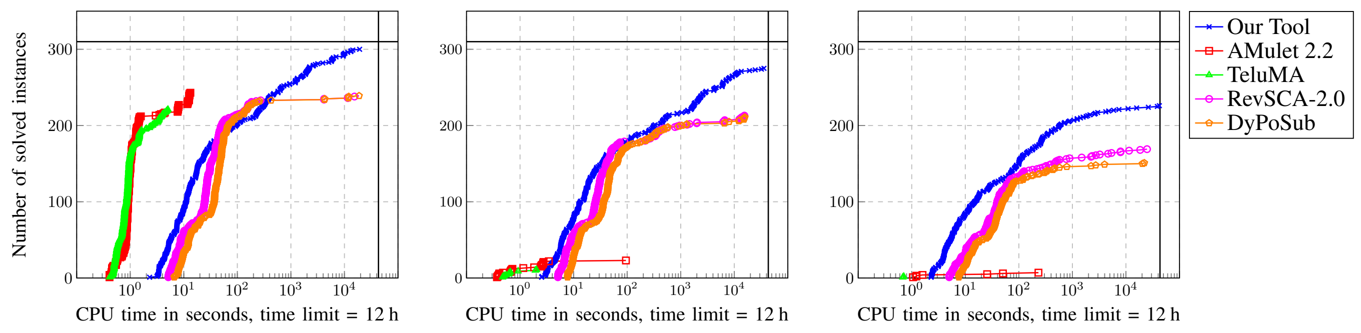
Certification is an important aspect to increase the trust in fully automatic tools. To this day, most SCA-based verification tools lack certification. Kaufmann et al. have made efforts to provide (easily checkable) certificates for different versions of their rewriting tools so far [20], [22], [25], [36]. They even showed in [36] unsoundness for an existing SCA-based tool by using fuzzing techniques. An alternative approach to guarantee the soundness of an automatic verification is to formally verify the verification tool itself. This approach is taken by Temel et al. [32], [33], [46] for the automatic verification of multipliers (which is not based on SCA, however). They verified the verification tool with the ACL2 theorem prover [47].

We want to highlight that the simplicity of our new approach facilitates certification, although our prototype tool does not yet produce certificates. Whereas the intensive search process for a good substitution order and for a good phase assignment for the intermediate polynomials may be expensive, it is easy to write out the final set of signals to virtually insert double inverters for phase optimization (see Sect. IV-A) as well as the finally computed substitution order. The substitutions are only performed at the gate level and not in a hierarchical manner and we do not use any properties derived by SAT or other techniques which would need a separate proof method. Thus, a simple dedicated and formally verified proof checker could be used whose memory requirements are limited by the memory requirements of the verifier or the certificate could be simply mapped to the existing practical algebraic calculus (PAC) format [48].

V. Experimental Results

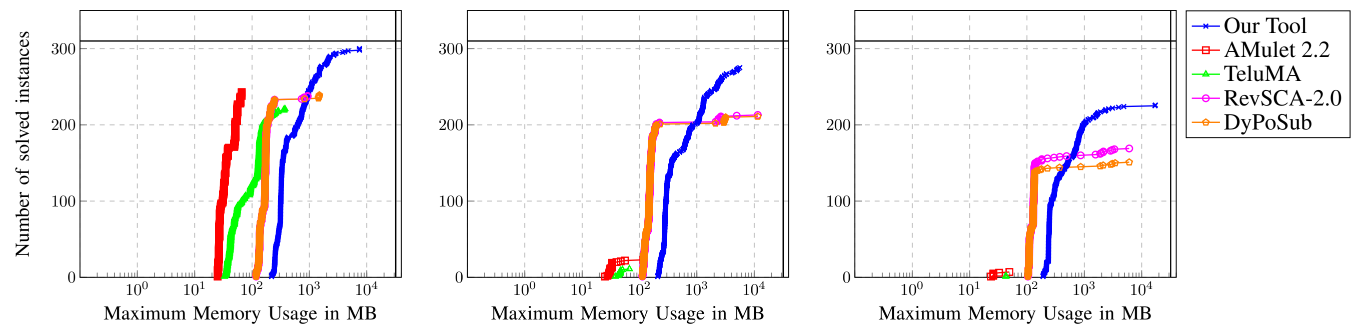
We have implemented the new method from this paper in our tool DynPhaseOrderOpt. Tests have been run on a single core of an Intel Xeon CPU E5-2650v2 with 2.60GHz. Resources were limited to 32GB main memory and 12 hours of CPU run time. For comparison we also run the following SCA-based multiplier verification tools on the same setup: AMulet 2.2 [25], [36], TeluMA [23], RevSCA-2.0 [24] and DyPoSub [21]. First experiments with the VeSCMul tool by Temel et al. [32], [33], [46] have shown that it is not suitable for flat AIG-based designs as we consider them here, since their rewriting method relies on hierarchy information and in case of flat AIG-based designs it fails even for multipliers with very small bitwidths. Therefore, the comparison with VeSCMul was omitted. The examined benchmark set contains 310 different unsigned 64-bit multiplier circuits and is composed of:

- *all* 192 64-bit unsigned multipliers from the aoki-benchmark set [49] (which unfortunately is no longer available online and therefore was obtained from the artifact data of [22])
- *all* 28 possible 64-bit unsigned multipliers obtainable from the multiplier generator GenMul [50], [51]
- *all* 90 possible 64-bit stand-alone unsigned multipliers obtainable from the multiplier generator multgen [52].



(a) No optimization. (b) Optimized with *resyn3*. (c) Optimized with *dc2*.

Fig. 3: Verification times for different tools and optimizations.



(a) No optimization. (b) Optimized with *resyn3*. (c) Optimized with *dc2*.

Fig. 4: Maximum memory usage for different tools and optimizations.

The multiplier circuits cover a wide range of different implementation options for PPGs, PPAs, and FSAs (see Sect. II-B). We provide the benchmark set, our tool and experimental data at [53].

The experimental results are shown in Tab. I. Col. 1 states the used tool. For each tool, we differentiate between five types of results (Col. 2): "Solved" means that the tool has successfully verified that the multiplier is correct within the given resource limits. "TO" indicates a time out, i.e. exceeding the time limit, while "MO" indicates a memory out, i.e. exceeding the available main memory. "SegFault" means the program was terminated by a segmentation fault and "F.Buggy" states that the multiplier circuit has been erroneously reported as buggy. For testing the robustness of the tools we consider different optimizations on the benchmark set given in Cols. 3 to 5 which are either *none* or the ABC [54], [55] commands *resyn3* and *dc2*. The numbers in Cols. 3 to 5 indicate how many results of each type a specific tool (indicated by the row) has produced for a given optimization variant (indicated by the column). In Col. 6 we sum up the results over all benchmarks. For AMULET 2.2 we first used its standard method which substitutes possible complex FSAs and verifies the modified circuits afterwards. Since AMULET 2.2 produced several segmentation faults when trying to substitute complex FSAs in *optimized* benchmarks, we have chosen the following approach: Whenever AMULET 2.2 produced a segmentation fault when trying to substitute complex FSAs, we omitted the substitution and ran the verification on the original circuit.

It can be seen that none of the tools is able to successfully verify all 310 unoptimized benchmarks (Col. 3). Our tool solves the most with 300 benchmarks, followed by AMULET 2.2 with 243, DYPOSUB with 239, REVSCA-2.0 with 238 and TELUMA with 221. The advantage of our tool can be seen even better in the results for optimized circuits. With logic optimization, we are still able to solve 275 benchmarks for *resyn3* and 226 for *dc2*. Here we also see the large deficit of AMULET 2.2 and TELUMA which can only solve up to 23 for any optimized benchmark set. While AMULET 2.2 runs into time outs for most instances, TELUMA also produces up to 143 segmentation faults for the optimized benchmarks. REVSCA-2.0 and DYPOSUB perform better on optimized benchmarks, but still lag behind our tool. They solve at least 62 benchmarks less for *resyn3* and 57 less for *dc2*. In total, our tool is able to solve 801 benchmarks while the second best tool, REVSCA-2.0, only solves 620 (Col. 6). In contrast to our tool, all comparison tools produce segmentation faults and even erroneously reported buggy results on some of the benchmark.

TABLE I: Verification results for different tools.

Tool	Result	none	resyn3	dc2	sum
		Optimization			
Our tool DYNPHASE-ORDEROPT	Solved	300	275	226	801
	TO	10	34	76	120
	MO	0	1	8	9
	SegFault	0	0	0	0
	F.Buggy	0	0	0	0
AMULET 2.2 [25], [36]	Solved	243	23	7	273
	TO	57	282	303	642
	MO	2	5	0	7
	SegFault	0	0	0	0
	F.Buggy	8	0	0	8
TELUMA [23]	Solved	221	11	2	234
	TO	89	219	165	473
	MO	0	1	0	1
	SegFault	0	77	143	220
	F.Buggy	0	2	0	2
REVSCA-2.0 [24]	Solved	238	213	169	620
	TO	21	20	53	94
	MO	21	70	73	164
	SegFault	12	0	0	12
	F.Buggy	18	7	15	40
DYPOSUB [21]	Solved	239	211	151	601
	TO	18	23	98	139
	MO	24	69	54	147
	SegFault	12	0	0	12
	F.Buggy	17	7	7	31

More details on the results are shown in Fig. 3 and Fig. 4 where we show cactus plots for the required run times and the maximum memory usage, respectively, for all tools and all optimizations (but only for solved instances). Fig. 3 shows that AMULET 2.2 and TELUMA are excellent wrt. time efficiency. All instances that could be solved needed 233 CPU seconds or less. A similar picture emerges for memory efficiency (Fig. 4). However, this advantage is paid by a much lower robustness; other tools show significantly better results for optimized benchmarks. This can be explained by the fact that AMULET 2.2 and TELUMA are tailored towards detecting certain structural pecularities in the circuit implementations. They are very fast, if those characteristics are found in the benchmarks. If logic synthesis has destroyed those structural properties, the other tools (and in particular our tool DYNPHASEORDEROPT) can demonstrate their robustness and their consistent performance for the general case.

In summary, the presented results show that our new tool DYNPHASEORDEROPT is not only able to solve almost all unoptimized benchmarks within reasonable times, but it also performs better than the other tools especially on optimized benchmarks, confirming the higher overall robustness of our method.

Finally, we investigated for our tool DYNPHASEORDEROPT the detailed question of whether it makes sense to try two precomputed orders based on breadth-first-search (BFS) and depth-first-search (DFS) first, before computing a dynamic order in Alg. 3 on the level of individual nodes within an EAB. The goal of this approach is to avoid time-intensive dynamic order computations for simple cases where BFS or DFS are sufficient. Tab. II clearly shows that the approach does make sense. For the table we consider all successful cases where the ordering for an EAB was not discarded later on by choosing another EAB to be processed before it on the higher "component level" of Alg. 2. We count how often BFS, DFS, and dynamic ordering was used, separately for each optimization (none, resyn3, dc2). Tab. II gives the fraction for each ordering method. It shows that most EABs are ordered by BFS. This fact may seem surprising at first sight, but can be explained by the fact that BFS is the default first order to try and it is chosen in all simple cases such as very small EABs with only a few nodes, EABs consisting of only one atomic block (like an XOR gate, an HA, or an FA which is anyway ordered according to a fixed precomputed order), or EABs which are just not very sensitive to different rewriting orders. Even though the number of dynamic orders applied on the level of individual nodes within an EAB is only up to 0.05 %, it is still important to use dynamic ordering to avoid exponential blowups while rewriting the individual nodes of an EAB in cases where neither BFS nor DFS are successful, since even the occurrence of just one such EAB in the entire circuit can lead to a failed verification attempt. Moreover, note that the precomputed BFS and DFS orders are used only on the level of individual nodes, while on the higher level of ordering EABs (see Alg. 2) a dynamic approach is always used.

TABLE II: Statistic on successful orders used in Alg. 3.

Optimization	% BFS	% DFS	% Dynamic
none	97.63	2.32	0.05
resyn3	98.49	1.47	0.04
dc2	98.40	1.58	0.02

VI. CONCLUSIONS AND FUTURE WORK

We have discussed the latest SCA-based approaches to fully automatic verification of multiplier circuits and presented a new, particularly simple method for this task. The new method consists of two major contributions. The first is our *Phase Optimization* algorithm which dynamically adjusts the phases of variables in occurring polynomials to reduce intermediate polynomial sizes. The second is our backward rewriting with dynamic *Order Optimization*, which uses several heuristics to create a dynamic order for backward rewriting that keeps intermediate polynomial sizes as small as possible. Our experiments show that our simpler method does not only compete well with latest tools on clean benchmarks but also adds more robustness, e.g. for the verification of optimized circuits. We believe that our dynamic approaches will be crucial for the verification of multipliers as well as other arithmetic circuits in the future.

References

[1] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs J.*, vol. 20, no. 4, pp. 129—-135, 1995.

[2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

[3] J. R. Burch, "Using BDDs to verify multipliers," in *DAC*, 1991, pp. 408–412.

[4] J. P. M. Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *DATE*. IEEE Computer Society / ACM, 1999, pp. 145–149.

[5] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *DATE*. IEEE Computer Society, 2001, pp. 114–121.

[6] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC*, 1995, pp. 535–541.

[7] R. E. Bryant and Y. Chen, "Verification of arithmetic circuits using binary moment diagrams," *STTT*, vol. 3, no. 2, pp. 137–155, 2001.

[8] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *ICCAD*, 1995, pp. 78–82.

[9] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, 2013.

[10] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *CAV*, 2008, pp. 473–486.

[11] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[12] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.

[13] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.

[14] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *DATE*, 2016, pp. 1048–1053.

[15] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.

[16] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on And-Inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.

[17] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.

[18] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.

[19] ——, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

[20] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.

[21] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.

[22] D. Kaufmann and A. Biere, "AMulet 2.0 for verifying multiplier circuits," in *TACAS*. Springer, 2021, pp. 357–364.

[23] D. Kaufmann, P. Beame, A. Biere, and J. Nordström, "Adding dual variables to algebraic reasoning for gate-level multiplier verification," in *DATE*. IEEE, 2022.

[24] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal," *TCAD*, vol. 41, no. 5, pp. 1573–1586, 2022.

[25] D. Kaufmann and A. Biere, "Improving AMulet2 for verifying multiplier circuits using SAT solving and computer algebra," *STTT*, vol. 25, no. 2, pp. 133–144, 2023.

[26] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability," in *DAC*, 2022.

[27] A. Yasin, T. Su, S. Pillement, and M. J. Ciesielski, "Formal verification of integer dividers: Division by a constant," in *ISVLSI*, 2019, pp. 76–81.

[28] ——, "Functional verification of hardware dividers using algebraic model," in *VLSI-SoC*, 2019, pp. 257–262.

[29] C. Scholl and A. Konrad, "Symbolic computer algebra and SAT based information forwarding for fully automatic divider verification," in *DAC*, 2020.

[30] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don't care optimization," in *DATE*. IEEE, 2021, pp. 1110–1115.

[31] A. Konrad, C. Scholl, A. Mahzoon, D. Große, and R. Drechsler, "Divider verification using symbolic computer algebra and delayed don't care optimization," in *FMCAD*. IEEE, 2022, pp. 1–10.

[32] M. Temel, A. Slobodová, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *CAV*, 2020, pp. 485–507.

[33] M. Temel and W. A. Hunt, "Sound and automated verification of real-world RTL multipliers," in *FMCAD*. IEEE, 2021, pp. 53–62.

[34] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., 2001.

[35] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor, "Polynomial formal verification of multipliers," *Form Methods Syst. Des.*, vol. 22, no. 1, pp. 39–58, 2003.

[36] D. Kaufmann and A. Biere, "Fuzzing and delta debugging and-inverter graph verification tools," in *TAP@STAF*. Springer, 2022, pp. 69–88.

[37] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 01 1951.

[38] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Comp.*, vol. EC-13, pp. 14–17, 1964.

[39] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[40] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Computer*, vol. 22, no. 8, pp. 786–793, 1973.

[41] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Computer*, vol. 31, no. 3, pp. 260–264, 1982.

[42] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[43] A. A. R. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using gröbner bases," in *FMCAD*. IEEE, 2016, pp. 169–176.

[44] R. Drechsler, B. Becker, and S. Ruppertz, "K*BMDs: A new data structure for verification," in *European Design & Test Conf.* IEEE Computer Society, 1996, pp. 2–8.

[45] S. Höreth and R. Drechsler, "Dynamic minimization of word-level decision diagrams," in *DATE*. IEEE Computer Society, 1998, pp. 612–617.

[46] M. Temel, "Vescmul: Verified implementation of S-C-Rewriting for multiplier verification," in *TACAS*. Springer, 2024, pp. 340–349.

[47] W. Hunt, M. Kaufmann, J. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philos. Trans. R. Soc. A*, vol. 375, p. 20150399, 2017.

[48] D. Kaufmann, M. Fleury, and A. Biere, "The proof checkers pacheck and pastèque for the practical algebraic calculus," in *FMCAD*. IEEE, 2020, pp. 264–269.

[49] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Formal design of arithmetic circuits based on arithmetic description language," *IEICE Trans. Fundamentals*, vol. 89-A, pp. 3500–3509, 2006.

[50] A. Mahzoon, D. Große, and R. Drechsler, "GenMul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*, R. Drechsler and D. Große, Eds. Springer International Publishing, 2021, pp. 177–191.

[51] ——, "Genmul," 2023. [Online]. Available: https://ics.jku.at/research/sca-verification/genmul/

[52] M. Temel, "Fast multplier generator multgen," 2019. [Online]. Available: https://github.com/temelmertcan/multgen

[53] A. Konrad and C. Scholl, "Benchmarks, binaries and experimental data," 2024. [Online]. Available: https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=24

[54] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[55] "ABC: A system for sequential synthesis and verification," available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2019.

Formal Methods in Computer-Aided Design 2024

Combining Symbolic Execution with Predicate Abstraction and CEGAR

Martin Jonáš
Masaryk University
Brno, Czechia
martin.jonas@mail.muni.cz

Jan Strejček
Masaryk University
Brno, Czechia
strejcek@fi.muni.cz

Alberto Griggio
Fondazione Bruno Kessler
Trento, Italy
griggio@fbk.eu

***Abstract*—The paper presents a simple, yet effective program verification technique that combines symbolic execution with implicit predicate abstraction and CEGAR. The technique can prove correctness of many programs that are beyond the reach of the standard symbolic execution because their symbolic execution tree is prohibitively large or even infinite. The technique has been implemented in the software model checker KRATOS. Our experimental evaluation shows that it also decides correctness of some programs that were decided neither by the standard symbolic execution nor by IC3 with predicate abstraction (all implemented in KRATOS).**

***Index Terms*—Program verification, symbolic execution, predicate abstraction, CEGAR.**

I. Introduction

Symbolic execution [Kin76] is a powerful and popular technique for static program analysis. It consists in exploring the behaviours of the program by traversing its control flow graph one path at the time, accumulating the constraints visited during such traversal in formulas called *path conditions*, which are then checked with a constraint solver (e.g., a SAT or SMT solver) for feasibility. Symbolic execution has been applied effectively to different program analysis tasks, including automated test generation [Kin76], software verification [JNS11], input filtering [CCZ+07], program debugging [QRLV12], and program repair [NQRC13], [MYR16]. Although primarily aimed at finding feasible paths satisfying a desired condition (e.g., reaching a target location, or traversing a specific set of locations), symbolic execution can also be used to prove unreachability of some error locations, by exhaustively enumerating all the feasible paths. In practice, however, this often diverges, because the number of such paths in many programs is prohibitively large or infinite (a simple example is shown in Fig. 1).

In this paper, we present a simple technique for improving the effectiveness of symbolic execution at proving unreachability. The main idea is to integrate *implicit predicate abstraction* [JM07], [Ton09] in the enumeration of paths, so as to ensure that the (abstract) symbolic execution tree is always finite (for a given set of predicates). This is done by setting up *abstraction locations* covering all program loops, assigning to each such location a finite set of *abstraction predicates*, and then restricting the symbolic exploration to *abstract simple paths*, i.e., paths in which all abstract states can occur at most once. To better control when the abstraction is applied, we also assign to each abstraction location an *abstraction threshold* saying that the abstraction is not applied in this location before the number of occurrences of the location in the current path exceeds the threshold. This can help avoiding the imprecise abstraction for loops with a small number of iterations. We show how these ideas can be integrated in a standard symbolic execution algorithm and included in a standard CEGAR loop [CGJ+03] with little effort, and demonstrate its effectiveness by evaluating our implementation in the KRATOS [GJ23] software model checker on a benchmark set obtained from the latest *Competition on Software Verification* SV-COMP [Bey24]. In particular, our results show that the new technique significantly improves the peformance of the symbolic execution engine of KRATOS on safe benchmarks (i.e., where the error location is unreachable), and it also can prove correctness of some programs that could not be verified by the other compared engines of KRATOS (within the given resource bounds), thus contributing to its overall performance on the benchmark set.

Paper outline: The rest of the paper is organized as follows. We introduce general background notions in Section II and standard symbolic execution in Section III. Our combination of symbolic execution, implicit abstraction, and CEGAR is described in Section IV. We present experimental evaluation in Section V and discuss related work in Section VI. Finally, we draw conclusions and discuss future directions in Section VII.

II. Preliminaries

Logic: We work in the setting of standard first-order logic. We use the standard notions of theory, satisfiability, and validity of a formula. For each term t and an assignment μ to variables and possibly uninterpreted function and relation symbols, $\mu(t)$ denotes the result of the evaluation of t under μ. Similarly, for a formula φ, we denote as $\mu(\varphi)$ the result of the evaluation of φ under μ. If the formula evaluates to *true*, we say that μ is a model of φ and write $\mu \models \varphi$. We assume that we work over a theory whose quantifier-free fragment

M. Jonáš and J. Strejček have been supported by the Czech Science Foundation grant GA23-06506S. A. Griggio has been supported by the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU, and by the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_33

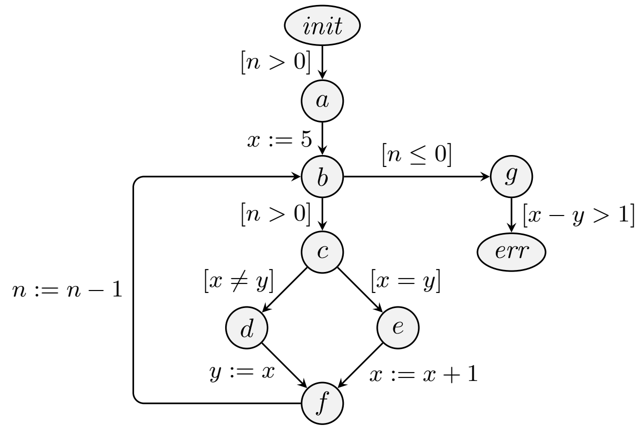


Fig. 1. A safe CFA where symbolic execution runs forever.

is decidable, i.e., there is a computable function $\text{IsSAT}(\varphi)$ that returns $true$ if φ is satisfiable and $false$ otherwise. For presentation purposes, all the examples in the paper are over the theory of linear integer arithmetic, but the concepts work for any theory with decidable quantifier-free fragment.

Mathematical notation: For each function $f \colon A \to B$ and $a \in A$, $b \in B$, we denote by $f[a \leftarrow b]$ the function that maps a to b and x to $f(x)$ for all x in $A \smallsetminus \{a\}$. The domain of a (partial) function f is denoted as $\operatorname{dom}(f)$. For each set A, we denote as A^+ the set of all non-empty sequences of elements from A. If $u, v \in A^+$, we denote their concatenation as $u.v$ (or just uv, if it is clear from the context). We denote the set of Booleans as $\mathbb{B} = \{true, false\}$.

Programs: We consider programs represented by *control-flow automata (*CFA*)*. Let $Vars$ be a fixed set of program variables. A control flow automaton is a tuple $A = (L, init, err, E)$, where L is a finite set of program locations, $init \in L$ is the initial location, $err \in L \smallsetminus \{init\}$ is the error location, and $E \subseteq L \times Ops \times (L \smallsetminus \{init\})$ is a finite set of edges between program locations that are labeled by operations. We assume that $init$ has only outgoing edges. Each operation $o \in Ops$ has one of the three following forms:

1) an *assumption* $[\varphi]$, where φ is a formula over $Vars$,
2) an *assignment* $x := t$, where $x \in Vars$ and t is a term over $Vars$, or
3) a *nondeterministic assignment* $x := *$, where $x \in Vars$.

We assume that if a single location has multiple outgoing edges, all of them are assumptions. A CFA used as a running example can be found in Fig. 1.

A *(control-flow) path* π leading to a location $l_k \in L$ is a nonempty finite sequence of consecutive edges $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k) \in E^+$ where $l_1 = init$. The path is called *error path* if $l_k = err$.

A program state is a pair (l, μ), where $l \in L$ is a program location and μ is an assignment of values to program variables. There is a transition $(l, \mu) \xrightarrow{(l,o,l')} (l', \mu')$ between two states along the edge $(l, o, l') \in E$ if one of the following holds:

1) $o = [\varphi]$, $\mu \models \varphi$, and $\mu = \mu'$, or
2) $o = (x := t)$ and $\mu' = \mu[x \leftarrow \mu(t)]$, or
3) $o = (x := *)$ and $\mu'(v) = \mu(v)$ for all $v \in Vars \smallsetminus \{x\}$.

A path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$ is *feasible* if there exists a sequence of assignments $\mu_1, \mu_2, \ldots, \mu_k$ satisfying $(l_i, \mu_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, \mu_{i+1})$ for all $1 \le i < k$. For example, the path $\pi = (init, [n > 0], a)(a, x := 5, b)$ in our running example is feasible, but the path $\pi.(b, [n \le 0], g)$ is not due to the contradictory assumptions on the value of n.

A CFA is called *unsafe* if there is a feasible error path and it is called *safe* otherwise. In the rest of the paper, we assume that $(L, init, err, E)$ is an arbitrary fixed CFA and we are interested in proving whether the CFA is safe or unsafe. We also assume that $Vars$ contains only the program variables used in the CFA.

III. Symbolic Execution

Symbolic execution [Kin76] is a technique that systematically explores all feasible paths of a given CFA. The main idea is that instead of concrete input values, symbolic execution uses variables representing arbitrary input values. Consequently, values of program variables are terms over the input variables. When symbolic execution evaluates an assumption $[\varphi]$, the program variables in φ are replaced by the corresponding terms and the resulting formula is added to the so-called *path condition*. The path condition is satisfiable if and only if the corresponding path is feasible. When the path condition becomes unsatisfiable, symbolic execution explores another path. Now we present symbolic execution formally.

Let $Inputs$ be a countably infinite set of variables that represent inputs of the program. A *symbolic state* is a pair (pc, m), where pc is a formula over $Inputs$ called a *path condition* and m is a *symbolic memory* which assigns to each program variable $x \in Vars$ a term $m(x)$ over $Inputs$ that represents the current value of x. We extend m to arbitrary terms and formulas. Namely, if t is a term over $Vars$, $m(t)$ is the result of simultaneously replacing each program variable x in t by $m(x)$. Analogously, $m(\varphi)$ is the formula over $Inputs$ obtained from a formula φ over $Vars$ in the same way. Given a symbolic state s, by $s.pc$ we denote its path condition and by $s.m$ its symbolic memory.

We assume that there is a function $fresh()$ whose every call returns a fresh variable from $Inputs$ and a function $freshMem()$ whose every call returns a symbolic memory that assigns to each program variable a fresh input variable.

We define a function $next$ that for each symbolic state s and each operation $o \in Ops$ returns the successor symbolic state $next(s, o)$. For a state $s = (pc, m)$ and an operation o, we set

$$next(s, o) = \begin{cases} (pc \wedge m(\varphi), m), & \text{if } o = [\varphi], \\ (pc, m[x \leftarrow m(t)]), & \text{if } o = (x := t), \\ (pc, m[x \leftarrow fresh()]), & \text{if } o = (x := *). \end{cases}$$

Algorithm 1 presents standard symbolic execution formulated as a recursive function exploring the tree of all feasible paths in a depth-first manner.

The function $\text{Symex}(l, s, h)$ has three arguments: the current location l, the current symbolic state s, and the sequence h

Algorithm 1 Standard symbolic execution

```
1:  function SYMEX($l, s, h$)
2:      $h \leftarrow h.(l, s)$                        ▷ update history
3:      if $l = err$ then
4:          return (UNSAFE, $h$)
5:
6:      for $(l, o, l') \in E$ do                  ▷ for each outgoing edge
7:          $s' \leftarrow next(s, o)$                     ▷ successor state
8:          if $o$ is an assumption then
9:              if not ISSAT($s'.pc$) then
10:                 continue                ▷ the path is not feasible
11:         $h' \leftarrow h.o$              ▷ new history with the operation
12:         if SYMEX($l', s', h'$) = (UNSAFE, $h''$) then
13:             return (UNSAFE, $h''$)        ▷ feasible error path
14:
15:     return SAFE       ▷ all outgoing feasible paths are safe
```

tracking the *history* of the current path (i.e., h stores the visited pairs of a location and a symbolic state interleaved with the performed operations). To symbolically execute a given CFA, we call SYMEX$(init, s_0, \varepsilon)$, where $s_0 = (true, freshMem())$ is a symbolic state with path condition $true$ and a fresh symbolic memory. The function first extends the history with (l, s). If the current location is err, then it returns UNSAFE and the current history representing the detected feasible error path. Otherwise, the function processes the edges leading from the current location l one by one. The operation of the edge is evaluated and if it changes the current path condition into an unsatisfiable one, the path is infeasible and we terminate its exploration. Otherwise, the operation is added to the history and symbolic execution is recursively called from the location and symbolic state after the operation. If this recursive call detects a feasible error path, the function produces the same verdict. If the recursive call finishes without finding any feasible error path, we continue with the next edge. If all edges are processed without finding any feasible error path, the function returns SAFE.

The biggest disadvantage of standard symbolic execution is its unability to show that a system with an infinite number of feasible paths is safe. This is, for example, the case of the CFA in Fig. 1: for each $j > 0$, the path going through locations $init.a(bcdfbcef)^j bg$ is feasible. Symbolic execution of such a path leads to the symbolic state with memory $m(n) = v_n - 2j$, $m(x) = 5 + j$, and $m(y) = 5 + (j - 1)$ and path condition equivalent to $v_n = 2j \wedge v_y \neq 5$, where $v_n, v_y \in Inputs$ represent the initial values of program variables n, y, respectively.

IV. Extending Symbolic Execution with Predicate Abstraction

One of the techniques used to reduce the number of states and paths of programs is *predicate abstraction* [BR02], [BHJM07]. Given a CFA A with program variables $Vars$ and a set $\mathbb{P}$ of formulas over $Vars$ called the *predicates*, the predicate abstraction is used to construct an abstract system $\widehat{A}_{\mathbb{P}}$ such that if the error state is unreachable in $\widehat{A}_{\mathbb{P}}$, it is also unreachable in the original system A. The system $\widehat{A}_{\mathbb{P}}$ has Boolean variables $Vars_{\mathbb{P}} = \{x_P \mid P \in \mathbb{P}\}$ that correspond to the predicates and its states are thus pairs $(l, \mu_{\mathbb{P}})$ of a location l and an assignment $\mu_{\mathbb{P}}\colon Vars_{\mathbb{P}} \to \mathbb{B}$ representing the current values of the predicates. There is a relation $H(\mu, \mu_{\mathbb{P}})$ between assignments to variables of the original and the abstracted system that holds if and only if $\mu(P) = \mu_{\mathbb{P}}(x_P)$ for all $P \in \mathbb{P}$. There is a transition $(l, \mu_{\mathbb{P}}) \xrightarrow{(l,o,l')} (l', \mu'_{\mathbb{P}})$ in $\widehat{A}_{\mathbb{P}}$ if and only if there exists a transition $(l, \mu) \xrightarrow{(l,o,l')} (l', \mu')$ in A such that $H(\mu, \mu_{\mathbb{P}})$ and $H(\mu', \mu'_{\mathbb{P}})$. The predicate abstraction can be further refined by assigning different sets of predicates to different program locations or by abstracting only in a subset of the locations [BKW10]. The computation of the transition relation in the abstract system is potentially expensive as it needs many SMT queries or alternative approaches with quantified SMT queries or AllSMT queries. This potentially expensive computation can be avoided by *implicit predicate abstraction* [JM07], [Ton09], where the abstraction itself is embedded in SMT queries asking for the existence of a certain path in the abstract system.

In the rest of this section, we present the main contribution of the paper, which is extending the symbolic execution with predicate abstraction and CEGAR. We do this in three conceptual steps. First, in Section IV-A we formalize the considered abstractions, define feasibility of paths in the abstract system, and introduce the *simplicity* of these paths which intuitively means that a path cannot pass the same abstract state twice. Section IV-B then presents our algorithm for symbolic execution extended with implicit predicate abstraction. Finally, Section IV-C incorporates the algorithm in a CEGAR loop that checks feasibility of the obtained abstract counterexamples and refines the abstraction.

A. Precision Function, Feasible and Simple Abstract Paths

First of all, we define *precision functions* that specify where, when, and what abstraction to use. Let $\mathcal{F}$ denote the set of formulas over program variables. A *precision function* is an arbitrary partial function $p\colon L \to \mathbb{N}_0 \times \mathcal{P}_{fin}(\mathcal{F})$ that assigns to a program location l a pair $p(l) = (c, \mathbb{P})$ of a non-negative integer c called *threshold* and a finite set $\mathbb{P}$ of *predicates*. Locations in $\mathrm{dom}(p)$ are called *abstraction locations* and the abstraction will be used only there. For an abstraction location l, the value $p(l) = (c, \mathbb{P})$ says that the abstraction in location l is applied only when the current path visits l at least c times and the abstraction uses the formulas of $\mathbb{P}$ as abstraction predicates. We refer to c and $\mathbb{P}$ assigned to l by p with $p(l).c$ and $p(l).\mathbb{P}$, respectively. In the following, we always assume that p denotes some precision function.

Given a path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$, we say that l_i is an *abstraction point* on π if it is an abstraction location that appears at least $p(l_i).c$ times in $l_1, l_2, \ldots, l_{i-1}$. Given an abstraction location l, we say that two assignments μ, μ' are *$p(l)$-equivalent* if they satisfy the same predicates assigned to l by p, i.e., for each $P \in p(l).\mathbb{P}$ it holds that

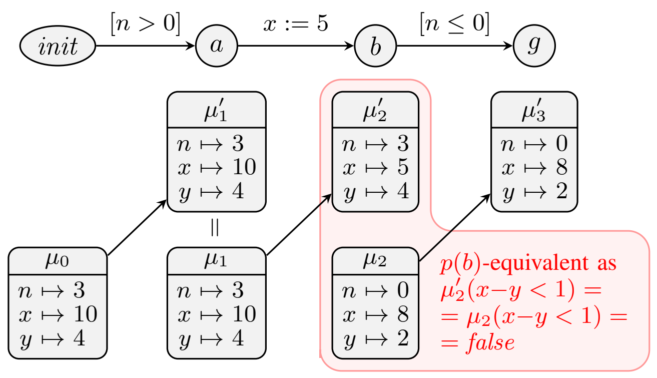


Fig. 2. A p-feasible path for $p(b) = (0, \{x - y < 1\})$ that is not feasible.

$\mu(P) = \mu'(P)$. The values $\mu(P)$ of the abstraction predicates $P \in p(l).\mathbb{P}$ form the *abstract state* associated to l.

Definition 1 (p-feasible path)**.** *We say that a control-flow path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$ is p-feasible if there exist assignment sequences $\mu_1, \mu_2, \ldots, \mu_{k-1}$ and $\mu_2', \mu_3', \ldots, \mu_k'$ such that for each edge (l_i, o_i, l_{i+1}) there is a transition $(l_i, \mu_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, \mu_{i+1}')$ and for each $1 < i < k$ it holds that if l_i is an abstraction point on π then μ_i, μ_i' are $p(l_i)$-equivalent and $\mu_i = \mu_i'$ otherwise.*

Theorem 1. *Each feasible path is also p-feasible for each precision function p.*

Proof. Let $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$ be a feasible path. As the path is feasible, there exist assignments $\mu_1, \mu_2, \ldots, \mu_k$ such that $(l_i, \mu_i) \xrightarrow{(l_i,o,l_{i+1})} (l_{i+1}, \mu_{i+1})$ for each $1 \leq i < k$. The path is also p-feasible as the assignment sequences $\mu_1, \mu_2, \ldots, \mu_{k-1}$ and $\mu_2' = \mu_2, \mu_3' = \mu_3, \ldots, \mu_k' = \mu_k$ clearly satisfy all the conditions in Definition 1. $\square$

Note that the other implication does not hold. For example, the path $(init, [n > 0], a)(a, x := 5, b)(b, [n \leq 0], g)$ of the running example is not feasible as mentioned in Section II, but Fig. 2 shows that it is p-feasible for precision function with $\mathrm{dom}(p) = \{b\}$ and $p(b) = (0, \{x - y > 0\})$.

It would not be useful to modify the symbolic execution to explore all p-feasible paths instead of all feasible paths as Theorem 1 implies that the number of p-feasible paths can only be higher. The key observation for our approach is that we do not have to explore *all* p-feasible paths, but only the *paths that do not contain two abstraction points with the same location and abstract state.* We call such paths p-simple. Formally, this is stated by the following definition and theorem.

Definition 2 (p-simple path)**.** *A control-flow path $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$ is called p-simple if it is p-feasible and there exist assignment sequences from the definition of p-feasibility that additionally satisfy the following: for all $1 \leq i < j \leq k$ such that l_i is an abstraction point on π and $l_i = l_j$ it holds that μ_i, μ_j' are **not** $p(l_i)$-equivalent.*

Theorem 2. *If there exists a feasible error path, then for each precision function p there is a p-simple error path.*

Proof. Let $\pi = (l_1, o_1, l_2)(l_2, o_2, l_3) \ldots (l_{k-1}, o_{k-1}, l_k)$ be a feasible path leading to $l_k = err$ and p be a precision function. Because π is feasible, there exist assignments $\nu_1, \nu_2, \ldots, \nu_k$ such that $(l_i, \nu_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, \nu_{i+1})$ for each $1 \leq i < k$. We show by induction that for each $1 < i \leq k$ there exists a path ρ leading to l_i and assignment sequences $\mu_1, \mu_2, \ldots, \mu_{|\rho|}$ and $\mu_2', \mu_3', \ldots, \mu_{|\rho|+1}'$ showing that ρ is p-simple and

1) $\nu_i = \mu_{|\rho|+1}'$ or
2) the last location on ρ is an abstraction point and $\nu_i, \mu_{|\rho|+1}'$ are $p(l_i)$-equivalent

Note that for $i = k$ this proves the statement.

Base case ($i = 2$) Consider the path $\rho = (l_1, o_1, l_2)$ and assignments $\mu_1 = \nu_1$ and $\mu_2' = \nu_2$. The path is p-feasible as $(l_1, \mu_1) \xrightarrow{(l_1,o_1,l_2)} (l_2, \mu_2')$. It is also p-simple as $l_1 = init$ has no incoming edges and thus $l_1 \neq l_2$.

Induction step ($i > 2$) The induction hypothesis gives us a path ρ' leading to l_{i-1} and assignment sequences $\mu_1, \mu_2, \ldots, \mu_{|\rho'|}$ and $\mu_2', \mu_3', \ldots, \mu_{|\rho'|+1}'$ showing that ρ' is p-simple and

1) $\nu_{i-1} = \mu_{|\rho'|+1}'$ or
2) the last location on ρ' is an abstraction point and $\nu_{i-1}, \mu_{|\rho'|+1}'$ are $p(l_{i-1})$-equivalent.

Consider the path $\rho'' = \rho'.(l_{i-1}, o_{i-1}, l_i)$ and the assignment sequences prolonged with $\mu_{|\rho'|+1} = \nu_{i-1}$ and $\mu_{|\rho'|+2}' = \nu_i$. Note that ρ'' is p-feasible as ρ' is p-simple, $(l_{i-1}, \mu_{|\rho'|+1}) \xrightarrow{(l_{i-1},o_{i-1},l_i)} (l_i, \mu_{|\rho'|+2}')$, and

1) $\mu_{|\rho'|+1} = \nu_{i-1} = \mu_{|\rho'|+1}'$ or
2) the last but one location on ρ'' is an abstraction point, and $\mu_{|\rho'|+1} = \nu_{i-1}, \mu_{|\rho'|+1}'$ are $p(l_{i-1})$-equivalent.

If ρ'' is also p-simple, we can simply set $\rho = \rho''$ as $\nu_i = \mu_{|\rho'|+2}' = \mu_{|\rho''|+1}'$.

If ρ'' is not p-simple, it has to be because of adding the last edge as ρ' is p-simple. Hence, there exists an abstraction point $l_{i'}$ on ρ' such that $l_{i'} = l_i$ and $\mu_{i'}, \mu_{|\rho''|+1}'$ are $p(l_i)$-equivalent. However, then we can set ρ to be the prefix of ρ' ending with $l_{i'}$. Such ρ leads to l_i, it is p-simple, its last location is an abstraction point and $\nu_i = \mu_{|\rho'|+2}' = \mu_{|\rho''|+1}', \mu_{i'} = \mu_{|\rho|+1}$ are $p(l_i)$-equivalent. $\square$

The important benefit of restricting the attention to p-simple paths is that for a suitable choice of the precision function p, there are only finitely many p-simple paths. In particular, we want to use a precision function p such that every cycle in the CFA contains at least one abstraction location. This is formalized by the following theorem, which will guarantee termination of the symbolic execution with predicate abstraction formulated in the next subsection.

Theorem 3. *Let p be a precision function such that each control-flow cycle contains at least one abstraction location $l \in \mathrm{dom}(p)$. Then the set of p-simple paths is finite.*

Algorithm 2 Symbolic execution with predicate abstraction

```
function SymExPA($l, s, h, p$)
    $h \leftarrow h.(l, s)$                                ▷ update history
    if $l = err$ then
        return (UNSAFE, $h$)

    if Abstract?($l, h, p$) then                   ▷ should we abstract?
        $m_A \leftarrow freshMem()$
        $pc_A \leftarrow s.pc \wedge eq(p(l).\mathbb{P}, s.m, m_A)$
        $pc_A \leftarrow pc_A \wedge simple(p(l).\mathbb{P}, l, m_A, h)$
        $s \leftarrow (pc_A, m_A)$

    for $(l, o, l') \in E$ do                      ▷ for each outgoing edge
        $s' \leftarrow next(s, o)$                     ▷ successor state
        if $o$ is an assumption or Abstract?($l, h, p$) then
            if not IsSAT($s'.pc$) then
                continue                       ▷ the path is not p-simple
        $h' \leftarrow h.o$               ▷ new history with the operation
        if SymExPA($l', s', h', p$) = (UNSAFE, $h''$) then
            return (UNSAFE, $h''$)         ▷ p-simple error path

    return SAFE        ▷ all outgoing p-simple paths are safe
```

Proof. We show that the length of each p-simple path is bounded from above by a constant. Let $L_A = \mathrm{dom}(p)$ be the set of abstraction locations, $L_N = L \smallsetminus L_A$ be the set of all non-abstraction locations, and π be a p-simple path. Since π is p-simple, it contains each abstraction location l at most $p(l).c + 2^{|p(l).\mathbb{P}|}$ times. Overall, it contains at most $b = \sum_{l \in L_A} p(l).c + 2^{|p(l).\mathbb{P}|}$ abstraction locations. Since each control-flow cycle contains at least one abstraction location, the path π does not contain more than $|L_N|$ consecutive locations from L_N. There are at most $b + 1$ consecutive segments of locations from L_N (initial, terminal, and between each two abstraction locations). The length of the path is thus at most $b + (b+1)|L_N|$. □

B. Symbolic Execution with Implicit Predicate Abstraction

The symbolic execution with predicate abstraction is computed by Algorithm 2. It is a modification of Algorithm 1 (the different parts are in red) that explores p-simple paths instead of feasible paths. In particular, Algorithm 2 builds path conditions that are satisfiable iff the corresponding path is p-simple.

The function Abstract?(l, h, p) returns *true* iff l is an abstraction location that has already been visited at least $p(l).c$ times by the current path (i.e., we are in an abstraction point). If this is not the case, the next step proceeds as in the standard symbolic execution. If we are in an abstraction point with a location l and a symbolic state s, we perform the abstraction before processing the next step. The abstraction resets the symbolic memory to a fresh memory m_A. To ensure that m_A represents assignments that are $p(l)$-equivalent with assignments represented by $s.m$, we add the formula $eq(p(l).\mathbb{P}, s.m, m_A)$ to the path condition, where

$$eq(\mathbb{P}, m, m') = \bigwedge_{P \in \mathbb{P}} (m(P) \leftrightarrow m'(P)).$$

To ensure p-simplicity, we add to the path condition the formula $simple(p(l).\mathbb{P}, l, m_A, h)$ satisfied by assignments where the memory m_A is not $p(l)$-equivalent with any memory previously visited by the path in an abstraction point with the same location l. Formally, we define

$$simple(\mathbb{P}, l, m, h) = \bigwedge_{\substack{(l', s') \in aPoints(h) \\ l' = l}} \neg eq(\mathbb{P}, m, s'.m)$$

where $aPoints(h)$ is the set of abstraction points and their corresponding symbolic states in the history h, i.e., $aPoints(h)$ contains the pairs (l', s') such that l' is an abstraction location appearing in h at least $p(l).c$ times before the pair (l', s').

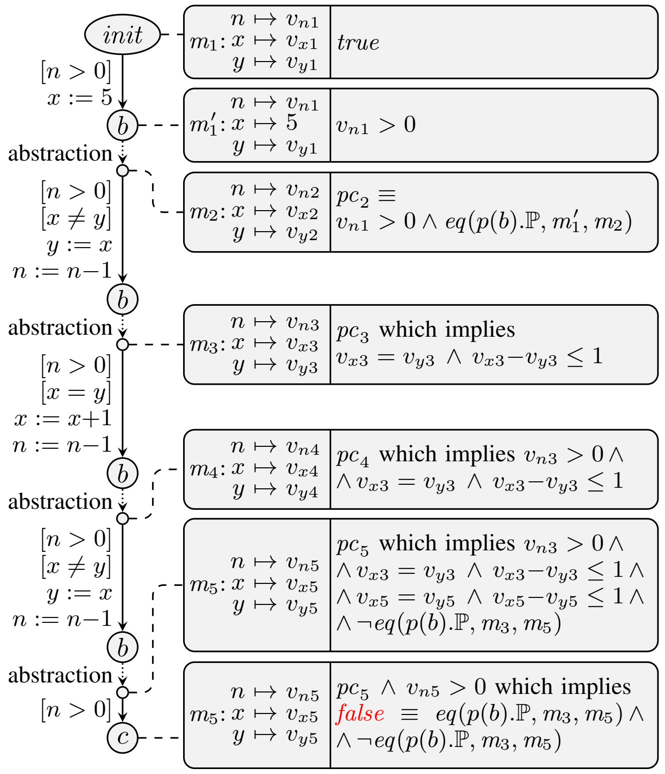
Theorem 2 implies that the algorithm is sound, i.e., if it returns SAFE, there is no feasible error path in the CFA. On the other hand, if the algorithm returns (UNSAFE, h), the p-simple error path represented by the history h can be infeasible. Theorem 3 implies that the algorithm terminates for each precision function p that defines at least one abstraction location on each control-flow cycle. This is true as there is only a finite number of p-simple paths and the algorithm checks satisfiability of the path condition that enforces p-simplicity one step after each abstraction point.

Algorithm 2 proves that our running example is correct if we use the precision function that specifies the only abstraction location b with $p(b) = (0, \{n > 0, x - y > 1, x = y\})$. We do not show the full computation due to space limits. Figure 3 sketches the computaiton along the path through locations $init.ab.cdfb.cefb.cdfb.c$ that ends with an unsatisfiable path condition meaning that the path is not p-simple. The figure fully presents the initial symbolic state, the symbolic state after the first two operations, and the symbolic state after the first abstraction (corresponding to the first values of m_A and pc_A in the algorithm). From the symbolic states after the next three abstractions and the final symbolic state we show only the symbolic memories and some imporant consequences of the path conditons.

C. Abstraction Refinement Loop

Algorithm SymExPA can be integrated into the standard CEGAR loop that checks the returned counterexamples for feasibility and iteratively refines the precision until SymExPA decides that the system is safe or a feasible error path is found. An implementation of this loop is presented in Algorithm 3. The algorithm uses three external functions:

- InitialPrecision() returns an initial set of abstraction locations with their thresholds and abstraction predicates, chosen either heuristically or by the user.
- IsFeasible(cex) checks feasibility of the path given by cex. This can be done by performing the standard symbolic execution along the path and checking satisfiability of the path condition.



init
m_1: $n \mapsto v_{n1}$, $x \mapsto v_{x1}$, $y \mapsto v_{y1}$
true
$[n > 0]$
$x := 5$
b
m'_1: $n \mapsto v_{n1}$, $x \mapsto 5$, $y \mapsto v_{y1}$
$v_{n1} > 0$
abstraction
$[n > 0]$
$[x \neq y]$
$y := x$
$n := n-1$
m_2: $n \mapsto v_{n2}$, $x \mapsto v_{x2}$, $y \mapsto v_{y2}$
$pc_2 \equiv$ $v_{n1} > 0 \wedge eq(p(b).\mathbb{P}, m'_1, m_2)$
b
abstraction
m_3: $n \mapsto v_{n3}$, $x \mapsto v_{x3}$, $y \mapsto v_{y3}$
pc_3 which implies $v_{x3} = v_{y3} \wedge v_{x3}-v_{y3} \leq 1$
$[n > 0]$
$[x = y]$
$x := x+1$
$n := n-1$
b
m_4: $n \mapsto v_{n4}$, $x \mapsto v_{x4}$, $y \mapsto v_{y4}$
pc_4 which implies $v_{n3} > 0 \wedge$ $\wedge\, v_{x3} = v_{y3} \wedge v_{x3}-v_{y3} \leq 1$
abstraction
$[n > 0]$
$[x \neq y]$
$y := x$
$n := n-1$
m_5: $n \mapsto v_{n5}$, $x \mapsto v_{x5}$, $y \mapsto v_{y5}$
pc_5 which implies $v_{n3} > 0 \wedge$ $\wedge\, v_{x3} = v_{y3} \wedge v_{x3}-v_{y3} \leq 1 \wedge$ $\wedge\, v_{x5} = v_{y5} \wedge v_{x5}-v_{y5} \leq 1 \wedge$ $\wedge \neg eq(p(b).\mathbb{P}, m_3, m_5)$
b
abstraction
$[n > 0]$
c
m_5: $n \mapsto v_{n5}$, $x \mapsto v_{x5}$, $y \mapsto v_{y5}$
$pc_5 \wedge v_{n5} > 0$ which implies $false \equiv eq(p(b).\mathbb{P}, m_3, m_5) \wedge$ $\wedge \neg eq(p(b).\mathbb{P}, m_3, m_5)$

Fig. 3. A sketch of the run of SYMEXPA($init$, ($true$, $freshMem()$), ε, p) on the path going through locations $init.ab.cdfb.cefb.cdfb.c$ with the precision function $p(b) = (0, \{n > 0, x - y > 1, x = y\})$

Algorithm 3 Symbolic execution with predicate abstraction and CEGAR

```
1: function SymexPA-CEGAR()
2:     $p \leftarrow$ InitialPrecision()
3:     $s \leftarrow (true, freshMemory())$
4:     while SymexPA($init, s, \varepsilon, p$) = (UNSAFE, $cex$) do
5:         if IsFeasible($cex$) then
6:             return (UNSAFE, $cex$)        ▷ real counterexample
7:         $p \leftarrow$ Refine($p$, $cex$)
8:     return SAFE                           ▷ abstract system is safe
```

- REFINE(p, cex) generates new predicates that block the spurious counterexample. Here, we treat it as a black-box that can be implemented by any existing technique for predicate generation. As a back-up solution for the case when predicate generation fails, the abstraction threshold can be increased for all locations on the path.

Similarly to BLAST [BHJM07], Algorithm 3 can be improved by not restarting the symbolic execution from scratch after a refinement. The symbolic execution can simply backtrack to the highest location whose precision was increased and restart from there with the new precision.

V. Experimental Evaluation

A. Implementation

We implemented the algorithm proposed in Section IV-C, including the symbolic execution backtracking after a refinement, in the KRATOS [GJ23] software model checker. The changes overall amounted to 778 lines of C++ code, including new user options related to the algorithm, logging, and statistics computation. The abstraction is performed only at loop heads. Refinement is implemented by computing sequence interpolants at the loop heads from the unsatisfiable feasibility query. The implementation relies on the SMT solver MathSAT5 [CGSS13] both for constraint solving and for interpolant computation. Because the proposed technique does not support function calls, the implementation first eagerly inlines all functions (and thus does not support unbounded recursion). Note that all engines of KRATOS, including the newly implemented one, support dynamic memory by modeling the heap and pointers using the theory of arrays.

The implementation is closed-source, but the binary is publicly available for academic and non-commercial use from https://www.fi.muni.cz/~xjonas/papers/fmcad24_symexecia/.

B. Experimental setup

For evaluation, we considered all the C programs from the *ReachSafety* category of the 2024 edition of the annual software verification competition SV-COMP [Bey24]. The category consists of 11 222 C programs divided into 15 benchmark families. We compare the standard symbolic execution implemented in KRATOS (*symexec*) and its proposed extension with implicit predicate abstraction and CEGAR using initial abstraction thresholds 0, 1, and 100 (*symexecia-0*, *symexecia-1*, *symexecia-100*, respectively). As external reference points, we execute the benchmarks using IC3 with implicit predicate abstraction [CGMT16] implemented in KRATOS (*IC3IA*), symbolic execution with CEGAR implemented in CPACHECKER [BL16] (*CPA-symexec+*), and finally SYMBIOTIC 10 [JKN$^+$24] (*Symbiotic*) as a well performing participant of SV-COMP based on the state-of-the-art symbolic executor KLEE [CDE08].

The experiments were performed on several identical PCs equipped with Intel Core i7-8700 CPU @ 3.20 GHz and 32 GiB of RAM. Each execution was limited to use a single CPU core, 5 minutes of wall time, and 8 GiB of RAM. For reliable benchmarking, all experiments were executed using BENCHEXEC [BLW19].

We observed that some of the tools produced *false positives*, i.e., returned *unsafe* for benchmarks marked as *safe*. In particular, *CPA-symexec+* has 44 false positives (23 in *Arrays*, 1 in *Fuzzle*, and 20 in *Heap*), *IC3IA* has 5 false positives (all in *Hardness*), *Symbiotic* has 1 false positive (in *Fuzzle*), *symexec* has 3 false positives (2 in *Hardness* and 1 in *Hardware*), and *symexecia-100* has 1 false positive (in *Hardware*). We do not consider these results in the rest of the evaluation and focus only on the correctly solved benchmarks.

C. Results

We first compare the results of the standard symbolic execution implemented in KRATOS with the proposed symbolic execution with predicate abstraction and CEGAR. The numbers of correctly solved benchmarks are shown in Table I.

TABLE I
NUMBERS OF CORRECTLY SOLVED UNSAFE (U) AND SAFE (S) BENCHMARKS BY STANDARD SYMBOLIC EXECUTION AND SYMBOLIC EXECUTION WITH PREDICATE ABSTRACTION AND CEGAR WITH VARIOUS INITIAL ABSTRACTION THRESHOLDS.

	Total		symexec		symexecia-0		symexecia-1		symexecia-100	
Family	U	S	U	S	U	S	U	S	U	S
Arrays	113	320	5	**10**	57	4	**59**	4	53	8
BitVectors	15	34	**11**	21	10	23	10	24	**11**	**27**
Combinations	430	241	**55**	**10**	0	2	0	2	4	7
ControlFlow	29	37	3	3	**4**	**15**	3	7	3	6
ECA	480	783	18	0	25	**51**	**29**	44	28	0
Floats	268	804	**39**	**202**	10	200	10	200	10	200
Fuzzle	0	15	0	0	0	0	0	0	0	0
Hardness	0	4005	0	824	0	**2315**	0	1646	0	833
Hardware	497	727	62	0	47	44	50	**49**	**71**	32
Heap	73	166	20	**52**	20	48	20	49	**21**	**52**
Loops	201	528	**114**	282	73	194	84	195	111	**286**
ProductLines	265	332	178	86	129	**156**	128	104	**213**	86
Recursive	54	102	0	0	0	0	0	0	0	0
Sequentialized	400	184	4	0	3	**4**	4	3	**6**	0
XCSP	59	60	**47**	**49**	**47**	**49**	**47**	**49**	**47**	**49**
Total	2884	8338	556	1539	425	**3105**	444	2376	**578**	1586

TABLE II
NUMBERS OF CORRECTLY SOLVED UNSAFE (U) AND SAFE (S) BENCHMARKS BY COMPETING TOOLS AND THE BEST CONFIGURATION OF SYMBOLIC EXECUTION WITH PREDICATE ABSTRACTION AND CEGAR.

	Total		CPA-symexec+		IC3IA		Symbiotic		symexecia-0	
Family	U	S	U	S	U	S	U	S	U	S
Arrays	113	320	68	1	66	1	**86**	**65**	57	4
BitVectors	15	34	11	11	12	**27**	**13**	16	10	23
Combinations	430	241	122	0	60	**24**	**211**	0	0	2
ControlFlow	29	37	9	**15**	4	**15**	**18**	4	4	**15**
ECA	480	783	38	279	145	**347**	**270**	0	25	51
Floats	268	804	**33**	156	31	78	21	**335**	10	200
Fuzzle	0	15	0	0	0	0	0	0	0	0
Hardness	0	4005	0	6	0	**3160**	0	98	0	2315
Hardware	497	727	38	9	**114**	**202**	48	0	47	44
Heap	73	166	50	26	20	46	**70**	**119**	20	48
Loops	201	528	110	110	57	130	**132**	**306**	73	194
ProductLines	265	332	128	271	249	**286**	**265**	94	129	156
Recursive	54	102	0	1	0	0	**50**	**44**	0	0
Sequentialized	400	184	82	7	7	10	**244**	**51**	3	4
XCSP	59	60	11	0	46	**50**	38	**50**	**47**	49
Total	2884	8338	700	892	811	**4376**	**1466**	1182	425	3105

The proposed technique significantly improves the number of decided *safe* benchmarks (1539 vs 3105 with initial threshold 0) and also slightly improves the number of decided *unsafe* benchmarks (556 vs 578 with initial threshold 100). The improvements occur among multiple benchmark families. Different initial abstraction thresholds provide different benefits and downsides (see for example *safe* benchmarks from *BitVectors* and *Loops* or *unsafe* benchmarks from *ProductLines*). Intuitively, the chosen thresholds determine the numbers of loop unrollings after which the abstraction is applied. Therefore, if some loops of the program need only a small number of iterations, a higher threshold allows exploring them precisely by the standard symbolic execution without applying the abstraction. The experiments show that this might be cheaper and beneficial in some cases.

Out of the 3530 benchmarks decided by *symexecia-0*, 2673 were decided without any refinements. Additionally, 149 benchmarks were decided after 1 refinement and required at most 8 predicates per abstraction location, 52 after 2 refinements with at most 28 predicates, 114 after 3 refinements with at most 14 predicates, 69 after 4 refinements with at most 101 predicates. The remaining 473 benchmarks required at least 5 refinements and at most 204 predicates per location.

Table II presents a comparison of the best-performing configuration of our algorithm, *symexecia-0*, with other competing tools. It can be seen that our algorithm outperforms other symbolic-execution-based competitors, *Symbiotic* and *CPA-symexec+*, on several families of *safe* benchmarks and also on some families of *unsafe* benchmarks. On the other hand, *symexecia-0* is outperformed by the other engine of KRATOS, *IC3IA*. However, we note that *symexecia* can be easily implemented into virtually any existing symbolic execution engine, whereas this is not the case of *IC3IA* as it uses a significantly different and more complex algorithm.

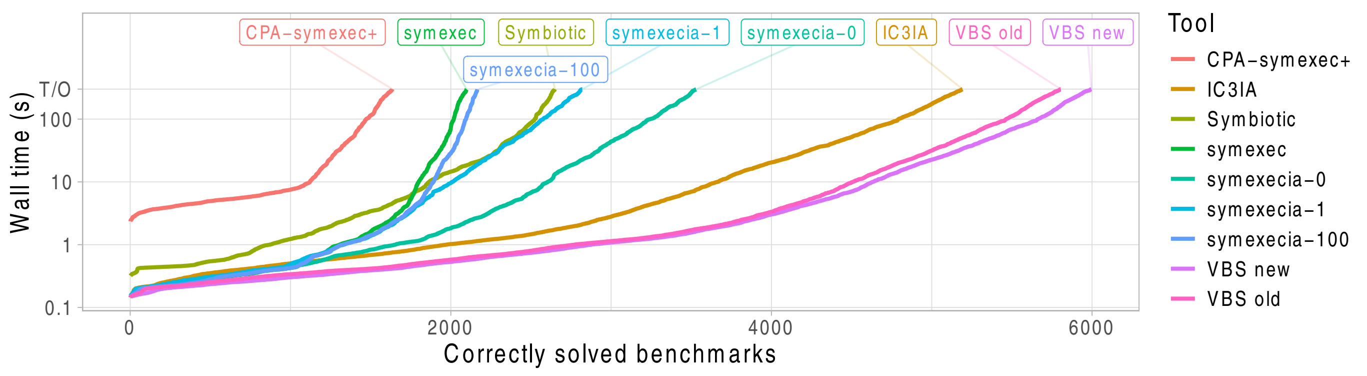


Fig. 4. The cactus plot of benchmarks solved by each tool. For each tool, the corresponding line shows the number of benchmarks (x-axis) solved in at most the given number of seconds of wall time (y-axis).

TABLE III
NUMBERS OF CORRECTLY SOLVED UNSAFE (U) AND SAFE (S) BENCHMARKS BY VIRTUAL-BEST SOLVER OF IC3IA + SYMEXEC IMPLEMENTED IN KRATOS AND THE VIRTUAL-BEST SOLVER OF IC3IA + SYMEXEC + ALL VARIANTS OF SYMEXECIA.

Family	VBS old		VBS new		Gain
	U	S	U	S	
Arrays	71	11	**72**	**12**	2
BitVectors	12	31	12	31	0
Combinations	68	26	68	**27**	1
ControlFlow	5	15	5	15	0
ECA	156	347	**159**	347	3
Floats	41	225	41	225	0
Fuzzle	0	0	0	0	0
Hardness	0	3208	0	**3369**	161
Hardware	134	202	134	202	0
Heap	22	52	22	52	0
Loops	127	358	**129**	**368**	12
ProductLines	258	315	258	**327**	12
Recursive	0	0	0	0	0
Sequentialized	7	10	7	10	0
XCSP	47	50	47	50	0
Total	948	4850	**954**	**5035**	191

To see whether the proposed technique brings any additional benefit to KRATOS compared to a simple parallel portfolio combination of predicate abstraction implemented in *IC3IA* and standard symbolic execution, we also compare the virtual-best solver composed of *IC3IA+symexec* (denoted as *VBS old*) and the virtual-best solver that also includes all variants of *symexecia-** (denoted as *VBS new*). The results are shown in Table III. *Symexecia* brings 6 newly solved *unsafe* benchmarks and 185 *safe* benchmarks across multiple benchmark families.

We also compared the runtime of all the tools. The number of solved benchmarks depending on the time-out is presented in the cactus plot in Figure 4. The plot supports all of the quantitative results from the tables and the previous paragraphs.

Additional plots and tables and all the logfiles from our experiments and scripts used for their analysis can be found at https://www.fi.muni.cz/~xjonas/papers/fmcad24_symexecia/.

Overall, despite its simplicity, our algorithm significantly outperforms symbolic-execution-based competitors on *safe* benchmarks and can solve benchmarks that can be solved neither by standard symbolic execution nor by more advanced approaches as IC3 with predicate abstraction.

VI. RELATED WORK

Our procedure can be seen as an instance of a more general family of techniques combining exploration of CFA paths with abstraction and refinement, using (lazy) predicate abstraction [BHJM07], [BKW10] and/or interpolants [McM06], [McM10], [BW12], possibly combined with symbolic execution and invariant inference [JNS11], [McM10]. All such approaches work by constructing abstract reachability graphs, in which nodes correspond to abstract states representing an overapproximation of states that are reachable by following some specific program paths, and rely on node coverage, i.e., showing that all the states represented by a given node n are contained within the states represented by a previously-visited node m, to ensure that the constructed abstract graph is finite. Our method, on the other hand, does not require the explicit computation of abstract states, and it relies only on (abstract) simple path constraints for making the abstract space finite. This is conceptually much simpler to integrate in a standard symbolic execution algorithm than approaches based on abstract states and covering such as [JNS11]; it should however be acknowledged that the technique of [JNS11] can potentially result in more compact abstract graphs.

There are other techniques that combine symbolic execution and abstraction, but in a different way and with a different aim than our procedure. For example, [APV09] extends symbolic execution with memory abstraction, but explicitly stores the visited abstract states, computes underapproximations of feasible paths, the abstract domain is fixed beforehand, there is no refinement, and the technique requires a dedicated algorithm for subsumption check. In [RMV09], the authors propose to use the abstract counterexample obtained by other means to guide the computation of standard symbolic execution on the original program towards the error location.

Another recent technique for combining symbolic execution with CEGAR and interpolation-based refinement is proposed in [BL16]. The difference with our approach is that in [BL16] the abstraction consists in tracking only a subset of the program variables and CFA constraints precisely (with interpolation-based refinement used to increase the set of variables and constraints to track), but the core symbolic execution algorithm is not modified; in particular, the technique does not guarantee that only finite abstract spaces are explored during each iteration of the CEGAR loop.

VII. Conclusions and Future Work

We presented a program verification technique that combines symbolic execution with implicit predicate abstraction and CEGAR, and implemented it the software model checker KRATOS. Our experimental evaluation showed that, despite its simplicity, the technique is effective in improving not only the proving capabilities of symbolic execution, but also the overall performance of KRATOS, by solving some benchmarks that could not be decided by its other verification engines.

As future work, we plan to extend the technique with interprocedural analysis, i.e., to handle function calls without relying on inlining. We also want to investigate additional ways of computing predicates during refinement, instead of relying on interpolation, and additional strategies for exploring the symbolic execution tree besides the current depth-first search.

References

[APV09] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.*, 11(1):53–67, 2009.

[Bey24] Dirk Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In *TACAS (3)*, volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *Int. J. Softw. Tools Technol. Transf.*, 9(5-6):505–525, 2007.

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *FMCAD*, pages 189–197. IEEE, 2010.

[BL16] Dirk Beyer and Thomas Lemberger. Symbolic execution with CEGAR. In *ISoLA (1)*, volume 9952 of *Lecture Notes in Computer Science*, pages 195–211, 2016.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In John Launchbury and John C. Mitchell, editors, *POPL*, pages 1–3. ACM, 2002.

[BW12] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. In *FMCAD*, pages 106–113. IEEE, 2012.

[CCZ$^+$07] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *OSR*, pages 117–130. ACM, 2007.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGMT16] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods Syst. Des.*, 49(3):190–218, 2016.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[GJ23] Alberto Griggio and Martin Jonáš. Kratos2: An SMT-based model checker for imperative programs. In *CAV (3)*, volume 13966 of *Lecture Notes in Computer Science*, pages 423–436. Springer, 2023.

[JKN$^+$24] Martin Jonáš, Kristián Kumor, Jakub Novák, Jindřich Sedláček, Marek Trtík, Lukáš Zaoral, Paulína Ayaziová, and Jan Strejček. Symbiotic 10: Lazy memory initialization and compact symbolic execution - (competition contribution). In *TACAS (3)*, volume 14572 of *Lecture Notes in Computer Science*, pages 406–411. Springer, 2024.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. *Log. Methods Comput. Sci.*, 3(4), 2007.

[JNS11] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded symbolic execution for program verification. In *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2011.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[McM10] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, pages 691–701. IEEE, 2016.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *ICSE*, pages 772–781. IEEE Press, 2013.

[QRLV12] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *TOSEM*, page 19, 2012.

[RMV09] Neha Rungta, Eric G. Mercer, and Willem Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In Corina S. Pasareanu, editor, *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*, volume 5578 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2009.

[Ton09] S. Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM*, pages 89–105, 2009.

Formal Methods in Computer-Aided Design 2024

Efficient Synthesis of Symbolic Distributed Protocols by Sketching

Derek Egolf
Northeastern University
Boston, MA USA
egolf.d@northeastern.edu

William Schultz
Northeastern University
Boston, MA
schultz.w@northeastern.edu

Stavros Tripakis
Northeastern University
Boston, MA
stavros@northeastern.edu

***Abstract*—We present a novel and efficient method for synthesis of parameterized distributed protocols by sketching. Our method is both syntax-guided and counterexample-guided, and utilizes a fast equivalence reduction technique that enables efficient completion of protocol sketches, often significantly reducing the search space of candidate completions by several orders of magnitude. To our knowledge, our tool, SCYTHE, is the first synthesis tool for the widely used specification language TLA+. We evaluate SCYTHE on a diverse benchmark of distributed protocols, demonstrating the ability to synthesize a large scale distributed Raft-based dynamic reconfiguration protocol beyond the scale of what existing synthesis techniques can handle.**

***Index Terms*—distributed protocols, synthesis, syntax-guided, counterexample-guided, sketching**

I. Introduction

Distributed protocols have become a crucial component in the operation of modern computer systems, including financial infrastructure [9], [8] and cloud data storage systems [10], [12]. In addition to being consequential and widely used, the complexity of these protocols makes them notoriously hard to design and reason about.

Automated verification of distributed protocols has made great advances in recent years. Specifically, inductive invariant inference methods have allowed for fuller automation of the verification of safety properties [18], [21], [50], [48], [41], [39]. State of the art tools in this domain are able to verify non-trivial specifications of *parameterized*, *infinite-state* protocols, written in languages such as TLA$^+$ [28] or Ivy [35]. Such verification efforts include not just protocol specifications especially designed to fit into the decidable fragment of Ivy [34], but also generally undecidable specifications of protocols such as Raft written in TLA$^+$ [41], [39] or Paxos written in Ivy [34], [19]. Progress is also being made towards fuller automation of the verification of liveness properties, e.g., see [49].

On the other hand, automated *synthesis* of distributed protocols is less advanced. This discrepancy might be expected because synthesis is intuitively a harder problem than verification: verification is about checking that a *given* system is correct, while synthesis involves inventing a system *and* ensuring that it is correct. Theory supports this intuition: model checking finite-state distributed systems is decidable, but synthesis of finite-state distributed systems is generally undecidable [37], [45], [46]. Synthesis of parameterized distributed protocols is also generally undecidable [25]. But even when decidable, synthesis "from scratch" is still a harder problem than verification, e.g., single-module reactive synthesis from LTL specifications is *doubly* exponential in the size of the LTL formula [36].

An easier problem than doing synthesis from scratch is to do synthesis by sketching [42], [43]. Sketching turns the synthesis problem into a completion problem: given a *sketch* (i.e., an incomplete system with *holes*) the goal is to complete the sketch such that the completion satisfies a given correctness specification. The holes are typically missing state variable updates, guards, or parts thereof. Completing a hole means finding the missing expression.

In this paper, we consider the problem of synthesis of distributed protocols by sketching. Contrary to prior works that either apply only to special classes of protocols [30], [24], or target protocols in an ad-hoc specification language [4], our work targets general protocols written in TLA$^+$ [28], a highly expressive specification language with widespread use in both academia and the industry [32].

Our approach follows the counterexample-guided inductive synthesis (CEGIS) paradigm [43], [20]: a *learner* is responsible for generating candidate solutions, while a *verifier* is responsible for checking whether a candidate satisfies the requirements.

Our synthesis method is truly *syntax-guided* in the sense that our synthesis loop explores directly the space of candidate symbolic expressions that can be generated from a given grammar. In contrast, previous work [4] explores the space of (finite) interpretations of uninterpreted functions. Our synthesis tool generates expressions, whereas the synthesis tool of [4] generates input-output tables (which can then be passed to an external SyGuS solver [1] to obtain an expression as a post-processing step). Our method does not rely on an external SyGuS solver.

A crucial component of our synthesis algorithm is how exactly we generate candidate expressions from a given grammar (or grammars, in the case of multiple holes). A naive, breadth-first enumeration of all possible expressions in the grammar

This material is partly supported by the National Science Foundation under Graduate Research Fellowship Grant #1938052, and Award #2319500. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_34

does not scale. Instead, we use a novel technique that employs *cache-based* enumeration coupled with an *equivalence reduction* with *short-circuiting*. This technique allows us to not only avoid checking semantically equivalent expressions, but also to avoid generating redundant expressions in the first place. Indeed, some of our experiments show a reduction in the number of generated expressions of more than three orders of magnitude.

We implement our method in a synthesis tool called SCYTHE. SCYTHE synthesizes protocols that are parameterized, i.e., in a form that is directly generalizable to an arbitrary number of nodes. Some of our synthesized protocols can be instantiated with infinite-domain variables, i.e., we can handle infinite-state protocols. SCYTHE is able to synthesize complex expressions from non-trivial grammars. For example, SCYTHE can synthesize the guard expression

$$\begin{aligned}&\forall Q_1 \in \mathit{Quorums}(\mathit{config}[i]),\\ &\quad \forall Q_2 \in \mathit{Quorums}(\mathit{new\_config}) : Q_1 \cap Q_2 \neq \emptyset\end{aligned} \tag{1}$$

which is parameterized by i, the node that is executing reconfiguration, and it can also synthesize the state variable update

$$\mathit{votes}' = [\mathit{votes} \text{ EXCEPT } [n_1] = \mathit{votes}[n_1] \cup \{n_2\}] \tag{2}$$

which is parameterized by n_1 and n_2, the nodes that are exchanging votes.

We evaluate SCYTHE on a suite of non-trivial benchmarks, including variants of the Raft dynamic reconfiguration protocol [33], [40], [41]. SCYTHE is able to synthesize correct and sometimes novel protocols in less than an hour and often in a matter of minutes. Although SCYTHE itself only guarantees correctness for a finite protocol instance, we were able to prove a-posteriori (using TLAPS [11]) that the synthesized protocols are in fact correct for an arbitrary number of nodes, as well as in some cases for infinite-domain state variables.

In summary, this work makes the following contributions: (1) A novel distributed protocol synthesis method that is both *syntax-guided* as well as *counterexample-guided*. (2) Novel techniques to accelerate the search of candidate completions, often reducing the search space by several orders of magnitude. (3) The synthesis tool SCYTHE which, to our knowledge, is the only tool able to handle a diverse suite of real-world distributed protocol benchmarks written in a broadly used language such as TLA+. (4) Formal correctness proofs which demonstrate that SCYTHE is able to synthesize infinite-state, parameterized protocols that are safe for any protocol instance.

II. PRELIMINARIES

A. Protocol Representation in TLA+

We consider symbolic transition systems modeled in TLA+ [28], e.g., as shown in Fig. 1. A primed variable, e.g., *vote_yes*′, denotes the value of the variable in the next state. Formally, a *protocol* is a tuple $\langle \text{PARAMS}, \text{VARS}, \text{INIT}, \text{NEXT} \rangle$. PARAMS is a set of *parameters* that may vary from one instantiation of the protocol to the other, but do not change during the execution of the protocol (e.g. a set of node ids *Node* in Fig. 1 line 1). VARS is the set of *state variables* (e.g. Fig. 1 line 2). INIT and NEXT are predicates specifying, respectively, the *initial states* and the *transition relation* of the system, as explained in detail in Sections II-B and II-D.

TLA+ is untyped, but for purposes of synthesis we assume that each symbol in PARAMS and VARS is typed. Supported types include *Bool* and *Int*, and sets or arrays of types. If *T1* and *T2* are types, then an element of type *(Set T1)* is a set of elements of type *T1* and an element of type *(Array T1 T2)* is a map from elements of type *T1* to elements of type *T2*.

A tuple $\langle \text{CONST}, \text{VARS}, \text{INIT}, \text{NEXT} \rangle$ denotes an *instance* of a protocol, where CONST is a mapping of PARAMS to values. For instance, $[\mathit{Node} \mapsto \{\mathit{n1}, \mathit{n2}, \mathit{n3}\}]$ characterizes one instance of the protocol in Fig. 1 and $[\mathit{Node} \mapsto \{\mathit{a0}, \mathit{b0}\}]$ characterizes another. The values of parameters need not be finite sets, e.g., *MaxVal* might have type *Int* and specify a bound on some value. Note that a protocol is technically not operational until the symbols in PARAMS are assigned to values, since the valuation of INIT and NEXT may depend on the valuation of the symbols in PARAMS.

A symbol in PARAMS may have type *Domain*, which designates it as an *opaque set*. If *Prm* is a PARAMS symbol of type *Domain* and CONST assigns *Prm* to set P, then an object x has type *(OfDomain Prm)* if and only if $x \in P$. For instance, if symbol *Node* in Fig. 1 has type *Domain*, then the symbol *vote_yes* has type *(Set (OfDomain Node))*. (Alternatively, *Node* could have type *(Set Int)*, but this typing would allow the protocol to, e.g., do arithmetic on node ids, which may not be desirable.) We discuss in Section III-B the use of opaque sets.

B. Protocol Semantics

A *state* of a protocol instance is an assignment of values to the variables in VARS. INIT is a predicate mapping a state to true or false; if a state satisfies INIT (if it maps to true), it is an initial state of the protocol.

The transition relation NEXT is a predicate mapping a pair of states to true or false. If a pair of states (s, t) satisfies NEXT, then there is a *transition* from s to t, and we write $s \rightarrow t$. A state is *reachable* if there exists a *run* of the protocol instance containing that state. A run of a protocol instance is a possibly infinite sequence of states $s_0, s_1, s_2 ...$ such that (1) s_0 satisfies INIT, (2) $s_i \rightarrow s_{i+1}$ for all $i \geq 0$, and (3) the sequence satisfies optional *fairness constraints*. We omit a detailed discussion of fairness. At a high-level, some transitions are called *fair* and under certain conditions, they must be taken. In this way, certain sequences of states are excluded from the set of runs of the protocol. In particular, if a sequence of states that would otherwise be a run ends in a certain cycle, that sequence may be excluded from the set of runs of a protocol due to fairness constraints.

C. Properties and Verification

We support standard temporal safety and liveness *properties* for specifying protocol correctness. Safety is often specified

```
1   CONSTANT Node
2   vars := (vote_yes, go_commit, go_abort)
3   GoCommit :=
4         ∧ vote_yes = Node
5         ∧ go_commit′ = Node
6         ∧ go_abort′ = go_abort
7   VoteYes(n) :=
8         ∧ vote_yes′ = vote_yes ∪ {n}
9         ∧ go_commit′ = go_commit
10        ∧ go_abort′ = go_abort
11  INIT :=
12        ∧ vote_yes = ∅
13        ∧ go_commit = ∅
14        ∧ go_abort = ∅
15  NEXT :=
16        ∨ GoCommit
17        ∨ ∃n ∈ Node : VoteYes(n)
```

Fig. 1. An example of a TLA+ protocol (excerpt).

using a state *invariant*: a predicate mapping a state to true or false. A protocol instance satisfies a state invariant if all reachable states satisfy the invariant. A protocol instance satisfies a temporal property if all runs (or fair runs, if fairness is assumed) satisfy the property. A protocol satisfies a property if all its protocol instances satisfy the property.

D. Modeling Conventions

We adopt standard conventions on the syntax used to represent protocols, particularly on how NEXT is written. Specifically, we decompose NEXT into a disjunction of *actions* (e.g. Fig. 1 lines 15-17). An action is a predicate mapping a pair of states to true or false; e.g., action *GoCommit* of Fig. 1. We decompose an action into the conjunction of a *pre-condition* and a *post-condition*. A pre-condition is a predicate mapping a state to true or false; if the pre-condition of an action is satisfied by a state, then we say the action is *enabled* at that state. For instance, Fig. 1 line 4 says that action *GoCommit* is enabled only when all nodes have voted yes.

We decompose a post-condition into a conjunction of *post-clauses*, one for each state variable. A post-clause determines how its associated state variable changes when the action is taken. For instance, Fig. 1 line 5 shows a post-clause for the state variable *go_commit*, denoted by priming the variable name: *go_commit*′.

In general, post-clauses may be arbitrary predicates involving the primed variable (e.g. $v' \in e$). We assume that all synthesized post-clauses are of the form $v' = e$ where e does not contain any primed variables, but make no assumptions on the post-clauses that we do not synthesize. In synthesis, non-determinism is used extensively, e.g., for modeling the environment. We note that the $v' = e$ assumption does not limit us to deterministic protocols, since multiple actions may be enabled at the same state.

Some actions are *parameterized*. For instance on line 7 of Figure 1, the action *VoteYes* is parameterized by a symbol n. From line 17, we can infer that n denotes an element of the set *Node*. The *arguments* of an action are those symbols like n. The *domain* of an argument to an action is the set quantified over for that argument in NEXT. For example, the domain of argument n of action *VoteYes* is the set *Node*. We require that the domain of an argument be a symbol from PARAMS of type *Domain*. An action may have multiple arguments; the domain of an action is the Cartesian product of the domains of its arguments. A parameterized action denotes a family of actions, one for each element in its domain.

If $s \to t$ is a transition and (s, t) satisfies an action A, we can say that A *is taken* and write $s \xrightarrow{A} t$. Note that (s, t) may satisfy multiple actions and we may annotate the transition with any of them. We write $s \xrightarrow{A(\vec{v})} t$ to explicitly denote the arguments to A; $\vec{v}$ is empty in the case of non-parameterized actions. In this way, runs of a protocol may be outfitted with a sequence of actions. Annotating runs of a protocol with actions is critical for our synthesis algorithm, since annotations allow us to "blame" particular actions for causing a counterexample run (c.f. Section IV-C). Fairness constraints are often specified using actions: we may say A *is (strongly) fair* to mean that action A must be taken if it is enabled infinitely often.

III. SYNTHESIS OF DISTRIBUTED PROTOCOLS

A. Protocol Sketches

A tuple $\langle \text{PARAMS}, \text{VARS}, \text{HOLES}, \text{INIT}, \text{NEXT}_0 \rangle$ is a *protocol sketch*, where PARAMS, VARS, and INIT are as in a TLA+ protocol and NEXT_0 is a transition relation predicate containing the hole names found in HOLES. HOLES is a (possibly empty) set of tuples, each containing a hole name h, a list of argument symbols $\vec{v}_h$, an output type t_h, and a grammar G_h. A hole represents an uninterpreted function of type t_h over the arguments $\vec{v}_h$. Each hole is associated with exactly one action A_h and it appears exactly once in that action. The grammar of a hole defines the set of candidate expressions that can fill the hole.

For example, a sketch can be derived from Fig. 1 by replacing the update of line 8 with $\mathit{vote\_yes}' = h(\mathit{vote\_yes}, n)$, where h is the hole name, the hole has arguments *vote_yes* and n, the return type is *(Set (OfDomain Node))*, and the action of the hole is *VoteYes*(n). One grammar for this hole might be (in Backus Normal Form):

$$E ::= \emptyset \mid \{n\} \mid \mathit{vote\_yes} \mid (E \cup E) \mid (E \cap E) \mid (E \setminus E)$$

which generates all standard set expressions over the empty set, the singleton set $\{n\}$, and the set *vote_yes*. We note that, in general, each hole of a sketch may have its own distinct grammar.

A hole is either a *pre-hole* or a *post-hole*. If the hole is a pre-hole, it is a placeholder for a pre-condition of the action. If the hole is a post-hole, it is a placeholder for the right-hand side of a post-clause of the action, e.g., as in $\mathit{vote\_yes}' = h(\mathit{vote\_yes}, n)$, where h is a post-hole. We do not consider synthesis of the initial state predicate and therefore no holes appear in INIT.

The arguments of a hole h may include any of the protocol parameters in PARAMS, the state variables in VARS, and the arguments of A_h if the action is parameterized. If h is a pre-hole, then its return type is boolean. If the hole is a post-hole, its type is the same as its associated variable, e.g., hole h above has the same type as *vote_yes*.

B. Problem Statements

A *completion* of a sketch is a protocol derived from the sketch by replacing each hole with an expression from its grammar. Informally, the synthesis task is to find a completion of the protocol that satisfies a given property. The distinction between a protocol and an instance of a protocol is important here; it may be easier to find a completion of a protocol such that a specific (e.g., finite) instance satisfies a property than to find a completion such that all instances satisfy the property. Therefore, we define two versions of the synthesis problem:

Problem 1. *Let* $\langle \textsc{Params}, \textsc{Vars}, \textsc{Holes}, \textsc{Init}, \textsc{Next}_0 \rangle$ *be a sketch and* Φ *a property. Let* $\textsc{Const}$ *be an assignment to* $\textsc{Params}$. *Find a completion,* $\langle \textsc{Params}, \textsc{Vars}, \textsc{Init}, \textsc{Next} \rangle$*, of the sketch such that the instance* $\langle \textsc{Const}, \textsc{Vars}, \textsc{Init}, \textsc{Next} \rangle$ *satisfies* Φ.

Problem 2. *Let* $\langle \textsc{Params}, \textsc{Vars}, \textsc{Holes}, \textsc{Init}, \textsc{Next}_0 \rangle$ *be a sketch and* Φ *a property. Find a completion,* $\langle \textsc{Params}, \textsc{Vars}, \textsc{Init}, \textsc{Next} \rangle$*, of the sketch such that every instance of the completion satisfies* Φ.

In this paper we focus on solving Problem 1. It is a more tractable problem and we are able to use a model checker as a subroutine in cases where the instance has finitely many states. It turns out that in many cases, a solution to Problem 1 generalizes and is also a solution to Problem 2. This generalizability comes from the fact that the symbols in PARAMS are opaque; e.g., we may refer to the set of nodes *Node*, but we cannot refer to any particular element of *Node* without quantification. Indeed, as we show in Section V, our tool is able to synthesize protocols that generalize, i.e., they are also solutions to Problem 2.

IV. Our Approach

As mentioned in the introduction, we follow the CEGIS paradigm which includes two main components: a learner and a verifier. In our case, the learner and verifier interact in a loop with the following steps: (1) the learner generates a candidate completion X, if one exists, that satisfies a (possibly empty) set of *pruning constraints* (i.e., X is pruned if it *violates* the constraints), (2) the verifier checks X against the supplied property Φ, (3) if X satisfies Φ, a solution is found and the algorithm terminates, (4) if X does not satisfy Φ, the verifier produces a counterexample run r, (5) the learner uses r to add new pruning constraints, and we repeat until a solution is found or the search space is exhausted.

Our learner component has three subcomponents: the *expression generator* (EG), the *pruning constraint checker* (PCC), and the *counterexample generalizer* (CXG). EG generates expressions from grammars, as detailed in Section IV-A. PCC checks each generated expression against the current set of pruning constraints, as explained in Section IV-B. CXG is invoked in Step (5) to update the pruning constraints by *generalizing* the information contained in the counterexamples, as detailed in Section IV-C.

Pruning constraints eliminate candidate completions that are guaranteed to exhibit previously encountered counterexamples, without having these candidates checked by the verifier, which is often an expensive subroutine. A naive way to do that would be to keep a list L of counterexamples seen so far, and then check whether a candidate exhibits any of the runs in L. Instead, we use more sophisticated pruning constraints that encode counterexamples as logical constraints on uninterpreted functions, c.f. Sections IV-B and IV-C.

As our verifier in Step (2), we use an off-the-shelf TLA$^+$ model checker, specifically TLC [51]. We will not discuss TLA$^+$ model checking further as it is standard.

A. Expression Generation

Recall that candidate protocols are completions of some sketch, which are in turn charaterized as members of some grammar. In the case of multi-hole sketches, completions are characterized as members of the cross-product of the grammars. Therefore, generating candidate protocols reduces to *enumerating* expressions from grammars. Note that, although grammars have a finite representation, the *language* (i.e., set of expressions) of a grammar may be infinite and the expressions therein may be arbitrarily large.

We experimented with three grammar enumeration techniques: (1) a naive breadth-first algorithm, (2) a cache-based algorithm, and (3) an extension of the cache-based algorithm that exploits semantic equivalence of expressions.

1) Naive Breadth-First Algorithm: A naive breadth-first search algorithm is to keep a priority queue (sorted by size) of *partial expressions*, i.e. expressions containing both terminals and non-terminals. A partial expression is discharged from the queue by considering all possible ways to replace the non-terminals using the grammar rules and substituting those into the partial expression. For example, if the partial expression is $d := E \cup (x \cup E)$ and the grammar has a production $G := E ::= x \mid y \mid E \cup E$, we would consider nine different partial expressions, one for each pair of productions of the E rule, since E appears twice in the expression d. After substituting, we can immediately return those expressions which do not contain non-terminals and add to the queue those that do. Our experience was that this algorithm was too slow in practice, since it iterates over and performs substitutions on larger and larger partial expressions.

2) Cache-Based Algorithm: In the cache-based algorithm, our learner generates all candidates of size n before it generates any candidates of size $n + 1$. Expressions are essentially trees and our notion of *size* is the number of nodes in the tree, e.g., the size of $(a + b) + c$ is 5. We keep a cache mapping each integer n to the set of all non-partial expressions of that size, for each non-terminal. We then use this cache to build larger non-partial expressions, substituting only into productions (partial expressions) that appear in the grammar. There are often many expressions of size n. We use generators to yield a stream of expressions, which avoids generating all expressions of a given size at once.

As an example of the cache-based algorithm, suppose we want to generate the expressions of size 5 for the non-terminal E in the grammar G above. Assume we already have a cache containing all expressions of size 1,2,3, and 4 for E. Then we can generate all expressions of size 5 by substituting pairs of expressions into the rule $E ::= E \cup E$ such that the sum of the sizes of the two expressions is 5.

3) Equivalence Reduction: Because the cache-based algorithm reuses all expressions of a given size many times over, it is important to keep the cache as small as possible. In particular, if two expressions are *semantically equivalent*, only one should appear in the cache. To illustrate, consider that there are only 16 boolean expressions over two variables, modulo equivalence. The grammar $B ::= x \mid y \mid \neg B \mid B \wedge B$ can express all 16 of these expressions, but it generates infinitely many expressions. The number of expressions of size n is $O(2^n)$.

When we generate a new expression, we compute a normal form for that expression. We then check if we have already generated an expression with that normal form. If we have, we do not return the new expression and we do not add it to the cache. We implement normal forms for (1) set expressions containing the operations $\cup$, $\cap$, and $\setminus$, (2) boolean expressions containing the operations $\vee$, $\wedge$, and $\neg$, (3) equality expressions, and (4) inequality expressions. We use DNF as the normal form for boolean expressions. Our normal form for set expressions exploits the correspondence between set expressions and boolean functions and then uses DNF. Equality between sets A and B is equivalent to $\emptyset = (A \setminus B) \cup (B \setminus A)$; we exploit this fact to obtain a normal form for equality between two sets.

In addition to equivalence reduction by normal forms, we also exploit the semantics of expressions to *short-circuit* the generation of expressions. Short-circuiting is a technique that allows us to avoid iterating over large parts of the search space. For instance, if we are generating expressions of size 5 for the rule $E \cup E$, we can consider pairs of expressions of sizes (1,4) and (2,3), but we can exploit the commutativity of union by ignoring sizes (4,1) and (3,2). Without this technique, we would have to iterate over twice as many pairs of expressions, compute their normal forms, and check if these normal forms are in the cache. In general, for commutative operation $\odot$ if we are generating expression $e_1 \odot e_2$, we first pick e_1 and only iterate over choices for e_2 that are at least as large as e_1.

B. Counterexamples and Pruning Constraints

1) Counterexamples: A counterexample is a run of the protocol annotated with actions: $s_0 \xrightarrow{A_1(\vec{v_1})} s_1 \xrightarrow{A_2(\vec{v_2})} \ldots \xrightarrow{A_k(\vec{v_k})} s_k$. The run is reported as a safety, deadlock, or liveness violation. If the run is a safety or deadlock violation, it is interpreted as a path. If it is a liveness violation, it is interpreted as a path leading to a cycle, called a *lasso*. In the case of liveness, $s_k = s_i$ for some $i < k$ and s_i is the state that first injects into the cycle.

2) Pruning Constraints: Our pruning constraints are logical constraints over propositional logic with equality and uninterpreted functions. For example, suppose we have the holes $h_1(a, b)$ and $h_2(b, c)$. Then, an example of a pruning constraint is the formula $\pi := (h_1(0, 1) \neq \textit{True}) \vee (h_2(1, 2) \neq 1)$. π constrains the candidate expressions for the holes h_1 and h_2. For example, replacing h_1 and h_2 with the expressions $a < b$ and $c - b$, respectively, violates π, because $0 < 1 = \textit{True}$ and $2-1 = 1$. Replacing h_1 and h_2 with $a < b$ and b, respectively, also violates π. Hence, π prunes at least two completions.

Formally, a pruning constraint is a disjunction of *terms*, where each term is a triple containing (1) a hole h, (2) a mapping $s^\star$ from the arguments of h to values, and (3) a literal value of the output type of h. For instance, in π above, the first term has $h = h_1(a, b)$, $s^\star = [a \mapsto 0, b \mapsto 1]$, and $y = \textit{True}$. The second term has $h = h_2(b, c)$, $s^\star = [b \mapsto 1, c \mapsto 2]$, and $y = 1$. Let $\tau := (h, s^\star, y)$ be a term and let $\widehat{h}$ be an interpretation (in our case, an expression) for the uninterpreted function h. Then $\widehat{h}$ satisfies τ if $\widehat{h}(s^\star) \neq y$. If $h_1, h_2, ..., h_m$ are the uninterpreted functions in a pruning constraint π and $X := \widehat{h}_1, \widehat{h}_2, ..., \widehat{h}_m$ are interpretations for the h_i (i.e. a completion), then X satisfies π if the disjunction of the τ terms in π is satisfied.

In each run, our algorithm maintains a *set* of pruning constraints, interpreted as a *conjunction* (of disjunctions of terms). A completion satisfies a set of pruning constraints if it satisfies all constraints in the set. Because we want to avoid seeing any counterexample more than once, the learner will pass a completion X to the verifier only if X satisfies every pruning constraint. I.e., a pruning constraint π prunes completions that do *not* satisfy π. PCC checks against the pruning constraints by substituting the expressions of the holes into the constraints and performing evaluation. Each type of counterexample (safety, deadlock, liveness) requires a slightly different encoding as a pruning constraint, as explained next.

C. Counterexample Generalization

A pruning constraint π is *under-pruning* w.r.t. run r and sketch S if there exists a completion X of S such that X satisfies π and r is a run of X. π is *over-pruning* w.r.t. run r and sketch S if there exists a completion X of S such that X does not satisfy π and r is not a run of X. π is *optimal* if it is neither under- nor over-pruning. π is *sub-optimal* if it is under-pruning, but not over-pruning. Our primary goal is to avoid over-pruning constraints, since over-pruning results in an incomplete algorithm, i.e., an algorithm that might miss valid completions.

In what follows we present three techniques to encode into pruning constraints, safety, deadlock, and liveness counterexamples, respectively. Our safety pruning constraints are optimal (Theorem 2), but our deadlock and liveness pruning constraints are sub-optimal (Theorems 3 and 4). In practice, these sub-optimal constraints are sufficient to avoid many completions that exhibit the corresponding violations; the bottleneck in our experiments is not the number of model checker calls.

1) Encoding Safety Counterexamples: Intuitively, a safety violation can be fixed by "cutting" at least one transition in the counterexample run, either by violating its guard or by modifying its state update. Let $r = s_0 \xrightarrow{A_1(\vec{v_1})} s_1 \xrightarrow{A_2(\vec{v_2})} \ldots \xrightarrow{A_k(\vec{v_k})} s_k$ be a safety violation and suppose that the completion is characterized by the interpretations $\widehat{h}_1, \widehat{h}_2, \ldots \widehat{h}_m$. We denote the pruning constraint for r as $\pi_{\mathit{safe}}(r)$ and construct it as follows. $\pi_{\mathit{safe}}(r)$ is a disjunction of *τ-terms*. For each $s \xrightarrow{A(\vec{v})} t$ in the counterexample, we construct a set of τ-terms. In particular, for each hole h_i in the action A, we construct the term $\tau_{A(\vec{v}),i} := (h_i, s^\star, y)$, where $y := \widehat{h}_i(s^*)$ and where $s^\star$ is the predecessor state s, restricted to the arguments of h_i, including the arguments to the action A. The pruning constraint is then the disjunction containing all $\tau_{A(\vec{v}),i}$.

For instance, suppose the safety violation is $[a, b, c \mapsto 0, 1, 2] \xrightarrow{A} [a, b, c \mapsto 1, 1, 2]$. Suppose additionally that $h_1(a, b)$ is a pre-hole in A and $a' = h_2(b, c)$ is a post-hole in A. Suppose that the completion that resulted in the safety violation had $\widehat{h}_1(0, 1) = \mathit{True}$ and $\widehat{h}_2(1, 2) = 1$. Then $\tau_{A,1} = (h_1, [a \mapsto 0, b \mapsto 1], \mathit{True})$ and $\tau_{A,2} = (h_2, [b \mapsto 1, c \mapsto 2], 1)$. The pruning constraint would be $\tau_{A,1} \vee \tau_{A,2}$, which corresponds to π from before. This constraint ensures that the precondition of A is not satisfied in the state $[a, b, c \mapsto 0, 1, 2]$ or that $a \neq 1$ after taking action A in that state.

2) Encoding Deadlock Counterexamples: Informally, a pruning constraint of a deadlock violation is similar to that of a safety violation because a deadlock violation can be fixed by making the deadlocked state s_k unreachable. But another way to fix a deadlock violation is to make s_k *undeadlocked*, which may be done by weakening the pre-condition of some action that is not enabled in s_k.

Formally, the deadlock pruning constraint for run r is defined to be $\pi_{\mathit{dead}}(r) := \pi_{\mathit{safe}}(r) \vee \pi_\rho(r)$, where $\pi_\rho(r)$ is a disjunction of *ρ-terms*, each of the form $\rho_{A(\vec{v}),i,k} := (h_i, s_k^\star, y)$, where $s_k^\star$ is s_k restricted to the arguments of h_i and where $y := \widehat{h}_i(s_k^\star)$. We construct a ρ-term for every action A and every pre-hole h_i in A such that $\widehat{h}_i(s_k) = \mathit{False}$. Then $\pi_\rho(r)$ is the disjunction of all all ρ-terms.

3) Encoding Liveness Counterexamples: The constraint for a liveness violation can be thought of as a generalization of the constraint for a deadlock violation. It is sufficient to do one of (1) break the path to the cycle using τ-terms, (2) break the cycle using τ-terms, or (3) weaken the pre-condition of some fair action that is not enabled in some state of the cycle using ρ-terms, making the cycle *unfair*. Formally, we denote our liveness pruning constraint as $\pi_{\mathit{live}}(r)$. We construct it as $\pi_{\mathit{live}}(r) := \pi_{\mathit{safe}}(r) \vee \pi'_\rho$, where π'_ρ is the disjunction of the following ρ-terms: For each fair action A, for every $\vec{v}$ in the domain of A, for every j such that s_j is in the cycle, and for every *pre-hole* h_i in A such that $\widehat{h}_i(s_j) = \mathit{False}$, we construct the term $\rho_{A(\vec{v}),i,j}$.

4) Fairness and Stuttering: Although we are able to handle both weakly and strongly fair actions, we did not treat them differently above in π_{live}. That construction may be under-pruning in the presence of weakly fair actions, but it will never over-prune and therefore our algorithm is complete. None of our benchmarks required weak fairness when modeling the synthesized protocols.

Stuttering (a special liveness violation) occurs when there are no fair, enabled, non-self-looping actions in the final state of the violation. In constrast, deadlock violations occur when there is no enabled action at all. We denote the pruning constraint for a stuttering violation as $\pi_{\mathit{stut}}(r) := \pi_{\mathit{safe}}(r) \vee \pi_\tau \vee \pi'_\rho$. In addition to the τ-terms from $\pi_{\mathit{safe}}(r)$, we add π_τ, which is the disjunction of $\tau_{A(\vec{v}),i}$ for every post-hole h_i in every fair action A. We add π'_ρ as we did for a typical liveness violation, except the only s_j in the cycle is the last state of r, s_k.

Theorem 1. *Let r be a counterexample of a completion of the sketch S. If r is a safety violation then $\pi_{\mathit{safe}}(r)$ is optimal w.r.t. r and S. If r is a deadlock, liveness, or stuttering violation then $\pi_{\mathit{dead}}(r)$, $\pi_{\mathit{live}}(r)$, and $\pi_{\mathit{stut}}(r)$, respectively, are sub-optimal w.r.t. r and S.* — The proof can be found in Appendix A.

V. Implementation and Evaluation

Implementation and Experimental Setup: We implemented our method (Section IV) in a tool, SCYTHE, which supports many features of the TLA$^+$ language and utilizes the TLC model checker [51] as verifier. SCYTHE is written in Python and takes as input (1) a TLA+ file defining the protocol and its sketch and (2) a configuration file defining the grammars and types along with protocol parameters. Our grammars are typed regular tree grammars [15] and our implementation essentially uses the standard SYNTH-LIB input format for SyGuS [1]. We ran each experiment on a dedicated 2.40 GHz CPU.

Benchmarks: Our benchmark suite contains seven distinct protocols: (1) decentralized lock service (decentr. lock), (2) server-client lock service (lock_serv), (3) Peterson's algorithm for mutual exclusion, (4) two phase commit (2PC), (5) consensus, (6) sharded key-value store (sharded_kv), (7) raft-reconfig, and (8) raft-reconfig-big. (7) and (8) are non-trivial, reconfigurable variants of the Raft protocol [33], [40], [41]. Our benchmarks are adapted from safety verification benchmarks that have been used in recent years [41], [17]. These existing benchmarks contain a suite of correct, manually crafted protocols and we refer to each manually crafted solution as the *ground truth*. We report statistics about the ground truth for reference, but we do not use this information during synthesis. For instance, we do not assume knowledge of which variables a missing expression depends on.

Adapting verification benchmarks for synthesis by sketching requires a number of steps, some of which are non-trivial. We discuss the most salient points of these steps next.

Holes: For each protocol we performed many synthesis experiments by varying the number of holes in the protocol sketch. All our experiments, as well as instructions for reproducing them, can be found on GitHub [13]. Representative experiments are summarized in Table I, explained below.

Grammars: Each hole requires a grammar. SCYTHE is flexible; the user can provide a different grammar for each hole, or reuse grammars across holes. SCYTHE grammars are *modular* in the sense that they contain a *general-purpose* part (e.g., the grammar of boolean or arithmetic or set expressions) plus a *hole-specific* part (e.g., the terminals which are the hole's arguments). We implemented a library that allows to build grammars by (1) automatically constructing non-terminals based on the types of the hole's arguments and (2) exposing to the user common sub-grammars that can be deployed across protocols.

Liveness and Fairness: Our benchmarks come from existing suites focusing on *safety* verification [41], [17]. Performing synthesis against only safety properties often results in *vacuous* solutions that satisfy safety in trivial ways (e.g. by filling a pre-hole with the expression *False*). Therefore, we augment each benchmark with additional liveness properties and any necessary fairness constraints.

Implementability Constraints: In addition to excluding vacuous solutions by adding extra properties, we sometimes need to exclude *unimplementable* solutions, for instance, solutions violating implicit communication/observability constraints between the protocol processes. For example, replacing the post-condition in line 8 of Fig. 1 with $\mathit{vote\_yes}' = \emptyset$ results in an unimplementable protocol because a node cannot directly change the vote state of another node. To avoid such solutions, we used arrays instead of sets (e.g., *vote_yes* is an array mapping process ids to booleans). Then, we restricted the grammar to only contain array access expressions with appropriate indices.

Explicitly Modeling the Environment: We had to modify several of the verification benchmarks of [41] in order to explicitly separate the (controllable) protocol from its (uncontrollable) environment, so as to prevent synthesis of parts belonging to the environment.

Results: TLA$^+$ LOC is the number of lines of code of the ground truth TLA$^+$ protocol specification, which is the same as the lines of code in the sketch and the synthesized protocols, since all synthesized expressions are printed to one line, regardless of size. ID refers to the number used to identify the experiment in the full results table [13]. "#pre/post holes" is the number of pre- and post-holes in the sketch, and k refers to $k = k_1 + k_2 + ... + k_n$, where each k_i is the *size* of the expression (c.f. Section IV-A) used in the ground truth protocol for the ith hole. "gram. LOC" is the number of lines (non-boilerplate) code in the python script used to generate the grammar. Every protocol uses the same grammar generation script, regardless of which or how many holes are poked.

We report Execution Statistics for the tool with and without equivalence reduction. The column "generated / model checked" reports the number of completions generated by the tool vs those model checked (the rest were pruned). The column k' is either the size of the expression found by the tool, or the size of the largest expression the tool considered before it timed out (marked with a $\geq$ symbol). If there are multiple holes then k' is the sum of all expression sizes. Column "total / model checking time" reports the total execution time vs the time devoted to model checking (both in seconds). TO indicates that the tool timed out after 1 hour; TO** is explained below. Note that TLC is called without a timeout; hence, it performs exhaustive model checking on the finite protocol instances specified by the user configuration.

As Table I shows, our efficient expression generation technique with equivalence reduction achieves impressive results, sometimes reducing the number of generated expressions by more than three orders of magnitude (c.f. raft-reconfig ID 121 where only 271 expressions are generated with reduction, vs $>$ 690,000 without reduction at the TO point). In all cases, the number of completions model checked is much smaller than those generated, which shows how critical pruning constraints are to scalability. With equivalence reduction, execution time is typically dominated by model checking, although there are exceptions (e.g. 2pc). Without equivalence reduction, the time is typically dominated by expression generation, which demonstrates the importance of the equivalence reduction.

Qualitatively, SCYTHE often synthesizes large, non-trivial expressions, e.g., single expressions of size 14 in the cases of raft-reconfig ID 121 and raft-reconfig-big ID 714, and multiple expressions of combined size up to 18 in other cases. Expressions (1) and (2) shown in Section I are two concrete examples of synthesized expressions.

Novel Solutions: The protocols synthesized by SCYTHE were often identical (or almost identical, up to commutativity of an operator such as $\wedge$, etc.) to the ground truth. In other cases, however, SCYTHE found novel, non-vacuous solutions. SCYTHE often found solutions with shorter expressions. One notable example comes from the experiment 2pc ID 303, where instead of the ground-truth expression $e_1 := \emptyset \neq (P \setminus A) \cup (P \cap N)$, SCYTHE found the expression $e_2 := P \neq (A \setminus N)$, where P is the set of all nodes, A is the set of alive nodes, and N is the set of nodes that voted no. So e_1 says "There is a node that is dead or there is at least one node that voted no." In the context of the protocol, e_1 and e_2 are equivalent, but a proof requires the subtle reasoning that both A and N are subsets of P, since P is the set of all nodes.

Correctness of Infinite Instances: A protocol synthesized by SCYTHE is a solution to Problem 1, i.e., is correct for the finite instance specified by the user. This correctness follows from the fact that during the synthesis loop the verifier (TLC) exhausts the state space of the specified finite instance. As it turns out, the protocols of Table I produced by SCYTHE are also solutions to Problem 2. Specifically, for each protocol of Table I except peterson, we used the TLA$^+$ Proof System (TLAPS) [11] to prove that all instances of that protocol satisfy the key safety property (our TLAPS proofs did not consider liveness; the four peterson variants involve only two processes and need no extra verification). For our TLAPS proofs we used techniques similar to those reported in [40].

In all but two cases, the initial solutions produced by SCYTHE proved to be correct. For raft-reconfig ID 343 and raft-reconfig-big ID 709, SCYTHE initially produced a solution which is correct for up to 3 nodes, but which we were surprised to find is incorrect for 4 or more nodes. To address this scenario, we added to the tool an *extra-check* option to perform an additional model checking step with larger parameter values

						Execution Stats					
		Sketch Parameters				w/o eq. reduction			w/ eq. reduction		
Protocol	TLA$^+$ LOC	ID	#pre/post holes	k	gram. LOC	**generated/** model checked	k'	**total** / model checking time	**generated/** model checked	k'	**total** / model checking time
decentr. lock	48	486	1 / 3	21	28	**93403** / 136	16	**674** / 193	**3020** / 117	*16*	**190** / 175
lock_serv	83	599	2 / 6	16	22	**384569** / 85	16	**2116** / 134	**7483** / 80	*16*	**159** / 120
lock_serv	83	611	2 / 6	16	22	**665463** / 104	$\geq$16	**TO** / 1094	**4064** / 84	*16*	**145** / 124
peterson	105	475	3 / 1	19	53	**442553** / 324	12	**2731** / 453	**485** / 243	*12*	**348** / 346
peterson	105	375	2 / 2	19	53	**583201** / 264	13	**3355** / 356	**1369** / 267	*13*	**364** / 357
peterson	105	413	3 / 1	22	53	**582616** / 353	$\geq$12	**TO** / 602	**7073** / 1259	*15*	**1809** / 1753
peterson	105	547	2 / 6	22	53	**643529** / 167	$\geq$16	**TO** / 483	**5569** / 222	*17*	**329** / 301
2pc	134	303	2 / 0	15	46	**690411** / 24	$\geq$8	**TO** / 1494	**65994** / 23	*9*	**388** / 43
2pc	134	558	3 / 5	18	46	**410675** / 87	14	**2301** / 179	**89027** / 88	*14*	**629** / 171
2pc	134	485	2 / 2	17	46	**681190** / 44	$\geq$10	**TO** / 492	**493492** / 55	*11*	**2654** / 110
2pc	134	513	2 / 6	18	46	**642178** / 162	$\geq$18	**TO** / 1641	**98009** / 211	*18*	**886** / 382
consensus	127	624	2 / 6	17	56	**550501** / 97	17	**3442** / 606	**9988** / 63	*17*	**427** / 375
consensus	127	550	2 / 6	22	56	**483126** / 162	$\geq$17	**TO** / 1116	**53994** / 286	*18*	**2291** / 2011
sharded_kv	112	302	1 / 1	13	44	**248298** / 13	13	**1325** / 49	**469** / 14	*13*	**59** / 57
sharded_kv	112	365	1 / 3	22	44	**611512** / 129	$\geq$16	**TO** / 941	**3149** / 149	*17*	**472** / 455
raft-reconfig	174	463	1 / 3	21	82	**64832** / 64	14	**462** / 128	**1958** / 65	*14*	**139** / 129
raft-reconfig	174	343	2 / 0	21	82	**608586** / 215	$\geq$11	**TO** / 484	**41411** / 251	*17*	**750** / 530
raft-reconfig	174	121	1 / 0	18	82	**694552** / 12	$\geq$12	**TO** / 3589	**271** / 13	*14*	**31** / 27
raft-reconfig-big	304	708	1 / 3	21	85	**67237** / 78	14	**1815** / 1465	**3155** / 84	*14*	**1668** / 1651
raft-reconfig-big	304	709	2 / 0	21	85	**2231397** / 220	$\geq$12	**TO**** / 3950	**282106** / 252	*17*	**TO**** / 12943
raft-reconfig-big	304	710	1 / 0	18	85	**658492** / 12	$\geq$12	**TO** / 3588	**1369** / 13	*14*	**368** / 359
raft-reconfig-big	304	714	1 / 7	25	85	**221648** / 101	18	**2662** / 1519	**6500** / 102	*18*	**1802** / 1768

TABLE I

than those used in the synthesis loop, right before outputting the final solution (if the extra-check fails, the tool continues to search for a solution). SCYTHE with extra-check found a correct (for all instances) solution for raft-reconfig ID 343 in 750 secs (this includes the time spent for extra-checks). SCYTHE with equivalence redution also found a correct (for all instances) solution for raft-reconfig-big ID 709, although it timed out after a total of four hours (TO**) while performing the final extra-check for 4 nodes—that single final extra-check took about 2 hours. For ID 709, SCYTHE without equivalence reduction failed to find a solution as it spent 4 hours generating expressions that were much smaller ($\leq$ size 12) than the solution found with equivalence reduction (size 17).

VI. Related Work

Past works synthesize explicit-state, finite-state machines [2], [5], [14], [16]. In contrast, we synthesize symbolic and parameterized infinite-state machines. TRANSIT [47] cannot process counterexamples automatically and requires a human in the synthesis loop. [30] and [6] use *cut-off* techniques which only apply to a special class of self-stabilizing protocols in symmetric networks, and [24] study a special class of distributed agreement-based systems. [29] consider only threshold-guarded distributed protocols. In contrast, our work applies to general distributed protocols.

As discussed in Section I, [4] synthesize interpretations of uninterpreted functions represented as finite lookup tables, whereas we synthesize symbolic expressions directly. We use TLA$^+$ models with parameterized actions. In contrast, [4] use extended finite state machines (EFSMs) which do not have parameterized actions. It is unclear whether expressions such as (1) and (2) on page 282 could be synthesized by [4].

Like [4], we use CEGIS and our counterexample encodings are similar. Unlike [4], we rely neither on an external SyGuS solver nor on an SMT solver. [4] encodes the search space of candidate interpretations as SMT formulas and calls an SMT solver to generate the next candidate. SMT queries are both expensive and numerous in the context of CEGIS. In contrast, we use efficient grammar enumeration techniques and we bypass SMT solvers by checking candidate expressions directly against the pruning constraints (Section IV).

Like [4], our tool synthesizes solutions that are guaranteed correct only up to the finite instances model checked in the CEGIS loop. Unlike [4], we went one step further and proved with TLAPS that the solutions produced by our tool are actually correct for all instances. As discussed in Section V, this step is not redundant: there were surprising cases of solutions which are correct for 3 nodes but not for 4 or more nodes. It is unclear whether the protocols synthesized in [4] are correct beyond the finite model checked instances.

[26] use genetic programming and [23] use machine learning for synthesis. Generally, these approaches are not guaranteed to find a solution even if one exists, i.e. they are incomplete. In contrast, our approach is complete.

None of the works cited above use syntax to guide the search, none use equivalence of expressions with short-circuiting to reduce the search space, and none handle state variables with infinite domains. To our knowledge, ours is the only truly syntax-guided synthesis method for symbolic, parameterized distributed protocols.

Existing SyGuS solvers use SMT formulas to express properties, and are therefore not directly applicable to distributed protocol synthesis which requires temporal logic properties. But our techniques for generating expressions and checking them against pruning constraints are generally related to term enumeration strategies used in SyGuS [1]. Both EUSolver [3] and cvc4sy [38] are SyGuS solvers that generate larger expressions from smaller expressions. EUSolver uses divide-and-conquer techniques in combination with decision tree learning and is quite different from our approach. To our knowledge, EUSolver does not employ equivalence reduction. The "fast term enumeration strategy" of cvc4sy is similar to our cache-based approach and also uses equivalence reduction techniques. To our knowledge, cvc4sy does not use short-circuiting.

In our work, we assume that the user has a means of constructing the appropriate sketch; we do not address the problem of "sketch inference." Work on scenarios [2] and flows [44] is directly applicable to this problem of coming up with a sketch. The sketch may also arise from a manually constructed, incorrect protocol that the user wishes to *repair* [7], along with knowledge of where a bug exists. Likewise, we assume that the user provided a sketch-property pair that is *realizable*—i.e., there exists a completion of the sketch that satisfies the property. Recent work on *unrealizability logic* [22], [27], [31] provides insight on how to identify unrealizable synthesis instances and communicate appropriate information to the user to help facilitate sketch debugging. A synthesis pipeline that integrates our work with that above is a promising direction for future work.

VII. Conclusion

We present the only, to our knowledge, truly syntax-guided synthesis method for symbolic, parameterized, infinite-state distributed protocols. We show experimentally that our method and tool are able to synthesize non-trivial completions across a broad set of non-trivial protocols written in TLA$^+$, and prove that these completions generalize correctly (i.e., preserve safety) in all possible instances.

Our sketch-based approach to distributed protocol synthesis is motivated by several factors. First, a common pattern in the design of distributed protocols is to extend an existing protocol (e.g. a non-reconfigurable protocol) with a new feature (e.g. reconfiguration). Indeed, our benchmarks include variants of the Raft dynamic reconfiguration protocol [33], [40], [41], and we focus on synthesizing the "Reconfig" action of those protocols. Sketching naturally fits this design pattern. Second, if a bug has been localized to a specific part of a protocol, sketching can be used to repair the protocol [7]. Finally, synthesis "from scratch" is a special case of synthesis by sketching where the sketch admits all protocols as completions. Therefore, no generality is lost when studying a sketch-based approach to synthesis and tractability is gained.

Future work includes: (1) further ways to reduce the search space and short-circuit parts of the search; (2) optimization of the SCYTHE-TLC interface to avoid running a new instance of (and repeatedly initializing) TLC each time SCYTHE needs to check a candidate protocol; (3) addressing the problems of sketch inference and unrealizability handling for the synthesis of distributed protocols; and (4) automating the final, all-instances verification step (generally an undecidable problem), by potentially combining TLAPS with state of the art inductive invariant inference techniques [39].

References

[1] 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.

[2] Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing Finite-state Protocols from Scenarios and Requirements. In *Haifa Verification Conference*, volume 8855 of *LNCS*. Springer, 2014.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[4] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic completion of distributed protocols with symmetry. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV*, volume 9207 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2015.

[5] Rajeev Alur and Stavros Tripakis. Automatic synthesis of distributed protocols. *SIGACT News*, 48(1):55–90, 2017.

[6] Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV*, volume 9779 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2016.

[7] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Automated model repair for distributed programs. *SIGACT News*, 43(2):85–107, 2012.

[8] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. 2016.

[9] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[11] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernan Vanzetto. TLA+ Proofs. *18th International Symposium on Formal Methods (FM 2012)*, 7436:147–154, January 2012.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[13] Derek Egolf. scythe-fmcad2024. https://github.com/egolf-cs/scythe-fmcad2024.

[14] Derek Egolf and Stavros Tripakis. Synthesis of distributed protocols by enumeration modulo isomorphisms. In *ATVA 2023 - Part I*, Lecture Notes in Computer Science, pages 270–291. Springer, 2023.

[15] Joost Engelfriet. Tree automata and tree grammars. *CoRR*, abs/1510.02036, 2015.

[16] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):519–539, 2013.

[17] Aman Goel. IvyBench. https://github.com/aman-goel/ivybench, Accessed: 2024-04-22.

[18] Aman Goel and Karem Sakallah. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021*, page 131–150, 2021.

[19] Aman Goel and Karem A. Sakallah. Towards an automatic proof of lamport's paxos. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 112–122, 2021.

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[21] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.

[22] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In Alastair F. Donaldson and Emina

Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1128–1142. ACM, 2020.

[23] Yujie Hui, Drew Ripberger, Xiaoyi Lu, and Yang Wang. Learning distributed protocols with zero knowledge. In *Machine Learning for Systems at NeurIPS 2023*, 2023.

[24] Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. Synthesis of distributed agreement-based systems with efficiently-decidable verification. In *TACAS 2023*, volume 13994 of *Lecture Notes in Computer Science*, pages 289–308. Springer, 2023.

[25] Swen Jacobs and Roderick Bloem. Parameterized synthesis. *Log. Methods Comput. Sci.*, 10(1), 2014.

[26] Gal Katz and Doron Peled. Synthesizing solutions to the leader election problem using model checking and genetic programming. In *Haifa Verification Conference*, HVC'09, page 117–132. Springer, 2009.

[27] Jinwoo Kim, Loris D'Antoni, and Thomas W. Reps. Unrealizability logic. *Proc. ACM Program. Lang.*, 7(POPL):659–688, 2023.

[28] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Jun 2002.

[29] Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *21st International Conference on Principles of Distributed Systems, OPODIS*, volume 95 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[30] Nahal Mirzaie, Fathiyeh Faghih, Swen Jacobs, and Borzoo Bonakdarpour. Parameterized synthesis of self-stabilizing protocols in symmetric networks. *Acta Informatica*, 57(1-2):271–304, 2020.

[31] Shaan Nagy, Jinwoo Kim, Loris D'Antoni, and Thomas W. Reps. Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. *CoRR*, abs/2401.13244, 2024.

[32] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.

[33] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319. USENIX Association, June 2014.

[34] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[35] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 614–630. ACM, 2016.

[36] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190, New York, NY, USA, 1989. Association for Computing Machinery.

[37] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[38] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Cesare Tinelli, and Clark Barrett. CVC4SY: Smart and fast term enumeration for syntax-guided synthesis. In Isil Dillig and Serdar Tasiran, editors, *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, volume 11561 of *Lecture Notes in Computer Science*, pages 74–83. Springer, July 2019.

[39] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing. arXiv eprint 2404.18048, 2024.

[40] William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 143–152. ACM, 2022.

[41] William Schultz, Ian Dardik, and Stavros Tripakis. Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA$^+$. In *22nd Formal Methods in Computer-Aided Design, FMCAD 2022*, pages 273–283. IEEE, 2022.

[42] Armando Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 4–13. Springer, 2009.

[43] Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, oct 2013.

[44] Murali Talupur, Sandip Ray, and John Erickson. Transaction flows and executable models: Formalization and analysis of message passing protocols. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, pages 168–175. IEEE, 2015.

[45] John G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005.

[46] Stavros Tripakis. Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1):21–28, April 2004.

[47] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296. ACM, 2013.

[48] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 485–501. USENIX Association, 2022.

[49] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. Mostly automated verification of liveness properties for distributed protocols with ranking functions. *Proceedings of the ACM on Programming Languages (POPL)*, 8:1028–1059, jan 2024.

[50] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021)*, pages 405–421. USENIX Association, July 2021.

[51] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

Appendix

A. Proof of Theorem 1

Theorem 1 follows from the four theorems below.

Theorem 2. *Let r be a safety violation of a completion of the sketch S. Then $\pi_{safe}(r)$ is optimal w.r.t. r and S.*

Proof. For brevity, $\pi := \pi_{safe}(r)$. To show that π is optimal, we must show that for any completion X of the sketch S, X satisfies π if and only if r is not a run of X. First suppose X satisfies π. Then π is not an empty disjunction and moreover there exists a $\tau_{A(\vec{v}),i}$ in π that is satisfied by X. This τ-term corresponds to some transition $s \xrightarrow{A(\vec{v})} t$ in r. If the h_i corresponding to the τ-term is a pre-hole, then the action $A(\vec{v})$ is disabled in state s of X. Suppose h_i is a post-hole corresponding to the state variable x. Then x has some value v_x in t. Because X satisfies π, we know that after taking action $A(\vec{v})$ in state s, x has some value $v_x^\star \neq v_x$. In either case, r is not a run of X because it cannot transition from s to t by action $A(\vec{v})$.

Now suppose X does not satisfy π. Then none of the τ-terms in π are satisfied. Let $T = s \xrightarrow{A(\vec{v})} t$ be a transition in r. We will show that X contains all such T and therefore r is a run of X. There are two cases: (1) the action of T has holes in it, or (2) it does not. In case (2), T is a transition that is present in every completion of the sketch. In case (1), we can leverage that X violates all $\tau_{A(\vec{v}),i}$ that were constructed for T. If it violates all $\tau_{A(\vec{v}),i}$ for all pre-holes, then $A(\vec{v})$ is enabled in state s. If it violates all $\tau_{A(\vec{v}),i}$ for all post-holes, then t is a successor of s by action $A(\vec{v})$ in X. $\square$

Theorem 3. *Let r be a deadlock violation of a completion of the sketch S. Then $\pi_{dead}(r)$ is sub-optimal w.r.t. r and S.*

Proof. Let $\pi := \pi_{dead}(r)$ for brevity. π is under-pruning because although ρ terms ensure that some pre-condition for some action A is weakened for deadlocked state s_k, it is possible that multiple pre-conditions need to be weakened in order for A to be taken in s_k.

Let X be a completion of the sketch S that does not satisfy π. To show that π is not over-pruning, we must show that r is a run of X and that the final state is deadlocked in X. As with the π_{safe} proof, we know that X has all the transitions in r, since all τ-terms are violated. Furthermore, we know that the final state of r is deadlocked in X because all ρ-terms are violated and therefore no pre-condition is weak enough to be taken in order to escape the deadlocked state. $\square$

Theorem 4. *Let r be a liveness violation of a completion of the sketch S. Then $\pi_{live}(r)$ is sub-optimal w.r.t. r and S.*

Proof. Let $\pi := \pi_{live}(r)$. As with the deadlock case, π is under-pruning because X satisfying π may only weaken one pre-condition of a fair action where it is necessary to weaken multiple pre-conditions to enable a fair action and make a cycle unfair.

Let X be a completion of the sketch S that does not satisfy π. Then all τ-terms are violated, so r is a run of X, so long as fairness constraints are satisfied. Fairness constraints are satisfied because all of the ρ-terms in π are violated. I.e., ρ-terms ensure there does not exist a fair action in X that is enabled in the cycle of r. $\square$

Theorem 5. *Let r be a stuttering violation of a completion of the sketch S. Then $\pi_{stut}(r)$ is sub-optimal w.r.t. r and S.*

Proof. Let $\pi := \pi_{stut}(r)$ and suppose X_0 is a completion of S that exhibits r. As with the deadlock and liveness violations, π is under-pruning because X satisfying π may only weaken one pre-condition of a fair action where it is necessary to weaken multiple pre-conditions to enable a fair action and make stuttering unfair.

Let X be a completion of the sketch S that does not satisfy π. We must show that r is a run of X. All terms of $\pi_{safe}(r)$ are violated, so the last state, s_k, of r is reachable in X by taking the sequence of transitions in r. Now, each fair action A of X is either enabled or disabled in s_k. We must show that all enabled fair actions are self-looping. Because the terms in π'_ρ are violated, we know that the non-self-looping fair actions that were disabled in state s_k of X_0 are also disabled in state s_k of X. Because the terms in π_τ are violated, we know that states that were self-looping in X_0 are still self-looping in X, if they are enabled in X. $\square$

Ownership in low-level intermediate representation

Siddharth Priya
University of Waterloo
Waterloo, Canada
siddharth.priya@uwaterloo.ca

Arie Gurfinkel
University of Waterloo
Waterloo, Canada
arie.gurfinkel@uwaterloo.ca

***Abstract*—The concept of ownership in high level languages can aid both the programmer and the compiler to reason about the validity of memory operations. Previously, ownership semantics has been used successfully in high level automatic program verification to model a reference to data by a first order logic (FOL) representation of data instead of maintaining an address map. However, ownership semantics is not used in low-level program verification. We have identified two challenges. First, ownership information is lost when a program is compiled to a low-level intermediate representation (e.g., in LLVM IR). Second, pointers in low-level programs point to bytes using an address map (e.g., in unsafe Rust) and thus the verification condition (VC) cannot always replace a pointer by its FOL abstraction. To remedy the situation, we develop ownership semantics for an LLVM-like low-level intermediate representation. Using these semantics, the VC can opportunistically model some memory accesses by a direct access of a pointer *cache* that stores byte representation of data. This scheme reduces instances where an address map must be maintained, especially for mostly safe programs that follow ownership semantics. For unsafe functionality, memory accesses are modelled by operations on an address map and we provide mechanisms to keep the address map and pointer cache in-sync. We implement these semantics in SeaBMC, a bit-precise bounded model checker for LLVM. For evaluation, the source programs are assumed to be written in C. Since C does not have ownership built-in, suitable macros are added that introduce and preserve ownership during translation to LLVM-like IR for verification. This approach is evaluated on mature open source C code. For both handcrafted benchmarks and practical programs, we observe a speedup of 1.3x–5x during SMT solving.**

I. Introduction

Ownership is a scheme to control aliasing of references in high level languages. It has been studied in a long line of academic research [1], [2], [3], [4]. More recently, the concept has gained attention due to Rust, a popular systems language that offers low level control like C/C++ and uses ownership semantics to record aliases and mutation of data. In Rust, (1) a value has exactly one owner, (2) a reference to a value (called a *borrow*) cannot outlive the owner, and (3) a value can have one mutable reference *xor* many immutable references. A program that follows this programming discipline allows the Rust compiler to reason about memory safety statically. However, for reasons of expressivity and performance, programs may need to break this discipline for certain operations. For this, Rust provides *unsafe* code blocks where the static checks are temporarily turned off.

While, ownership can aid in generating correct and efficient code, it is also useful in program verification. Usually, the presence of aliasing necessitates an *address map* to soundly model object accesses through different aliases. With ownership, this map can be eliminated when it is known that only a single reference exists. This has been useful for program verification. For example, the Move Prover [5] replaces references by objects in the generated verification conditions (VC). Similarly, RustHorn [3], [6] is able to generate pure First Order Logic VC for safe Rust programs without introducing a memory model.

The advances in verification using ownership semantics have not made their way to verification of low-level programs. One of the problems is that low-level languages do not support ownership out of the box. As an example, LLVM bitcode is a register based intermediate representation (IR) used by C, C++, and, Rust compilers. It only has an attribute for marking pointers as `noalias` [7] and no ownership operations. The `noalias` attribute is useful for optimization. However, the semantics of `noalias` in unclear and has caused confusion [8]. Another challenge is that ownership in high-level language does not translate directly to low-level settings. For example, in verification of safe Rust programs, it is correct to model a reference by the FOL representation of the value it refers to. However, this model does not work for LLVM-like IR (and unsafe Rust) because such languages (dialect) have pointers that treat values as a collection of bytes and rely on pointer arithmetic to access individual bytes. In verification, the standard solution models memory using an address map from addresses to byte or word values. However, such address maps are expensive to execute symbolically.

This work improves the state-of-the-art by the following contributions. First, we develop an ownership semantics for an LLVM-like low level language that operates on single words in memory. This language replaces unrestricted aliasing with mutable borrow, read-only borrow, and copy operations that track outstanding aliases for a memory allocation. Second, we define a caching mechanism for capturing data at a pointer itself. This cache can by written and read by operations on the pointer. A pointer cache allows us to replace memory accesses in the generated VC by the cache whenever correct to do so. This can simplify the VC and improve the solving time. For mostly safe programs, many memory accesses may be replaced by pointer cache accesses. These semantics are discussed in Section II.

Third, we discuss our design for VCGen and especially modelling the borrow operation in Section III. Borrowing temporarily transfers memory access rights from the lender pointer (a.k.a. *lender*) to the borrowing pointer (a.k.a. *borrower*). In

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_35

the given semantics, this means that the pointer cache is also copied from lender to borrower. However, the borrower is assumed to return the borrow by the first instance of a memory access by a lender. This means that the updated cache at the borrower *must* be copied back to the lender before this. This transfer ordinarily requires a memory model that allows shared accesses between the borrower and lender aliases. A more efficient way uses prophecy variables [9] and was first proposed in [3]. We adapt the prophecy solution to our setting.

Fourth, to make our ownership semantics practical, we add support for multi-word memory operations and evaluate the semantics by incorporating it in SEABMC [10]. SEABMC is a bit-precise bounded model checking engine for SEAHORN that uses an SMT solver as its backend. To ease writing programs using these ownership semantics, the user writes C programs laced with calls to ownership macros. During compilation, the macros expand to LLVM intrinsics that are then interpreted using the given semantics to generate VC. The benchmark programs are a mix of handcrafted examples and practical programs. The handcrafted examples are used to fine tune performance and show what is possible. The practical programs are from the `mbedTLS` project - an open-source SSL/TLS library and has routines for encryption and secure communications. We get a speedup of 1.3x–5x during SMT solving. We see that the verification simplicity (speedup) correlates positively with the number of memory accesses that can be replaced by pointer cache accesses in a program.

II. OSEA-IR LANGUAGE

In this section, we present the syntax and semantics of the OSEA-IR language. To simplify the presentation, we propose machines that work with a single datatype $bv(N)$, a bit-vector of N bits, as the *word* size. We impose two restrictions. First, all memory operations - *allocation*, *load*, and, *store* work on a single word. Second, the machine can only store integers in memory and does not support *store* and *load* of pointers. We lift the first restriction in Section IV, and the second in an extended version [11].

Syntax. We introduce ownership semantics on the base language SEA-IR [10]. SEA-IR is an intermediate representation (IR) itself based on LLVM IR. LLVM assumes a register based machine and dependency between memory operations are implied. SEA-IR explicates this dependency information between memory operations by introducing memory registers. We assume that the type of each register is known. Figure 1 shows the ownership extended syntax of SEA-IR called OSEA-IR. We use `R` to represent a scalar register, `P` for a pointer register and `M` for a memory register. A legal OSEA-IR program is assumed to be in a Static Single Assignment (SSA) form. OSEA-IR primarily replaces unrestricted alias creation by new operations that introduce and remove aliases in a restricted manner. The `mk_own` instruction initializes memory at the given location (simlar to a `Box::new(n)` in Rust). The `mut_mkbor`, `mut_mksuc` instructions occur in pairs. The first creates a mutable borrow pointer from a lender pointer. The second creates a succeeding pointer from the lender pointer that becomes active after the mutable borrow ends. The `mut_mkbor_off` is similar to a `mut_mkbor` and creates a pointer at an offset within an allocation. It must be followed by a `mut_mksuc` instruction. The `ro_*` instructions create read-only borrows of the lending pointer. The `cpy_*` instructions create unrestricted copies of the lending pointer. The `mut_mkbor_mem2reg` instruction borrows (loads) a pointer stored in memory to a register. The `mov_reg2mem` instruction moves (stores) a pointer in a register to memory. There is no move instruction between registers since the operation is equivalent to α-renaming.

Semantics of $\mathcal{M}_0$. The semantics are given in terms of a machine $\mathcal{M}_0$ and is based on the stacked borrows model for Rust [12]. In our formulation, each pointer type ($ptype$) is one of owned (o), mutably borrowed (mb), immutably borrowed (rb), or, copied (c). Access to memory is controlled by maintaining a per-location borrow stack that captures both valid accessors and access order. The configuration of $\mathcal{M}_0$ is given by the program counter state (P), register map (R : id $\rightarrow$ $value$), address map ($M : addr \rightarrow value$) and a borrow store state ($S_B : addr \rightarrow stack((tag, ptype))$). A $value$ is either a bit-vector $bv(N)$ or a pointer type. An address ($addr$) is represented as a bit-vector. A pointer is a tuple of $(addr, tag)$ and is considered *fat* on account of additional metadata carried along with the address[1]. A $tag : bv(N)$ is a unique id given to a pointer when it is defined. Operations that introduce and remove aliases, then push and pop alias tags on the borrow stack, respectively. Each borrow stack entry also stores $ptype$ along with an identifier for finer access control. An important restriction is that memory access is allowed for an alias if its tag is top-of-(borrow)stack for that address.

The semantics for relevant pointer introduction, aliasing, and, removal are given through operations on the borrow stack (B) in Table I. A borrow stack state is represented as a list $B = e :: B_1$, where e is the top of stack and B_1 represents the rest of the stack. We do not explicitly show effect of operations on (P, R, M) nor do we give the semantics for all instructions of $\mathcal{M}_0$ due to space constraints. The interested reader is referred to the stacked borrows [12] and SEA-IR [10] papers for further background. The `mk_own` operation allocates and stores n at the given location. This operation must provide a location that is un-allocated. After the operation, the new pointer tag is pushed onto the stack with $ptype =$ o. The mutable borrow operations use `mut_mkbor`, `mut_mksuc` instructions that always occur in a pair on a lender pointer p_0 to create a borrowed pointer q_0 and a succeeding pointer p_1. For a successful operation, the borrow stack is popped until p_0 is on top and its $ptype$ is either an owning or a mutably borrowed pointer. This operation removes $p_0.tag$ as an accessor and instead pushes $p_1.tag$, the succeeding pointer and $q_0.tag$, the borrowed pointer, to the borrow stack in that order. The associated type of a pointer is also added to each stack entry. Note that the type of q_1 is always mb. However

[1]We use the shorthand $.addr$ to refer to the first tuple element, similarly for other elements.

$\langle$S$\rangle$ ::= ... | $\langle$OS$\rangle$

$\langle$RDEF$\rangle$ ::= ... |
$\langle$P$\rangle$, $\langle$M$\rangle$ = `mk_own` $\langle$R$\rangle$, $\langle$M$\rangle$
$\langle$P$\rangle$ = `mut_mkbor` $\langle$P$\rangle$ | $\langle$P$\rangle$ = `mut_mkbor_off` $\langle$P$\rangle$, $\langle$R$\rangle$ | $\langle$P$\rangle$ = `mut_mksuc` $\langle$P$\rangle$ |
$\langle$P$\rangle$ = `ro_mkbor` $\langle$P$\rangle$ | $\langle$P$\rangle$ = `ro_mkbor_off` $\langle$P$\rangle$, $\langle$R$\rangle$ | $\langle$P$\rangle$ = `ro_mksuc` $\langle$P$\rangle$ |
$\langle$P$\rangle$ = `cpy_mkcpy1` $\langle$P$\rangle$ | $\langle$P$\rangle$ = `cpy_mkcpy1_off` $\langle$P$\rangle$, $\langle$R$\rangle$ | $\langle$P$\rangle$ = `cpy_mkcpy2` $\langle$P$\rangle$ |

$\langle$MDEF$\rangle$ ::= ... | $\langle$P$\rangle$, $\langle$M$\rangle$ = `mut_mkbor_mem2reg` $\langle$P$\rangle$, $\langle$M$\rangle$ | $\langle$M$\rangle$ = `mov_reg2mem` $\langle$P$\rangle$, $\langle$P$\rangle$, $\langle$M$\rangle$

$\langle$OS$\rangle$ ::= `die` $\langle$P$\rangle$

Fig. 1: Ownership instr. in OSEA-IR grammar, where `R`, `P`, and `M` are scalar registers, pointer registers, and memory registers respectively.

Operation	**Pre-condition**	**Post-condition**
`p,m1 = mkown n, m0`	$S_B[p.addr] = \varnothing$	$S_B[p.addr] = (tag_p, \mathsf{o}) :: []$

`q0 = mut_mkbor p0`
 `p1 = mut_mksuc p0`
	$S_B[p_0.addr] = B_0 :: (tag_{p_0}, t) :: B_1$,
	 $t \in \{\mathsf{o}, \mathsf{mb}\}, p_0.tag = tag_{p_0}$
		$S_B[p_0.addr] = (tag_{q_0}, \mathsf{mb}) :: (tag_{p_1}, t) :: B_1$

`c1 = cpy_mkcpy1 p0`
 `c2 = cpy_mkcpy2 p0`
	$S_B[p_0.addr] = B_0 :: (tag_{p_0}, t) :: B_1$,
	 $p_0.tag = tag_{p_0}$
		$S_B[p_0.addr] = (tag_{c_1}, \mathsf{c}) :: (tag_{c_2}, t) :: B_1$

`die q`	$S_B[q.addr] = (tag_q, t_q) :: (tag_p, t_p) :: B_1$, $q.tag = tag_q, t_q = \mathsf{mb}, t_p \in \{\mathsf{o}, \mathsf{mb}\}$	$S_B[q.addr] = (tag_p, t_p) :: B_1$
`m1 = store r, p, m0`	$S_B[p.addr] = B_0 :: (tag_p, t_p) :: B_1$, $t_p \neq \mathsf{rb}, p.tag = tag_p$	$S_B[p.addr] = B_2 :: (tag_p, t_p) :: B_1$, $(p.tag = tag_p, t_p \in \{\mathsf{o}, \mathsf{mb}\}) \implies (B_2 = \varnothing)$, $(t_p = c) \implies (B_2 = B_0)$
`r = load p, m`	$S_B[p.addr] = B_0 :: (tag_p, t_p) :: B_1$, $p.tag = tag_p$	$S_B[p.addr] = B_2 :: (tag_p, t_p) :: B_1$, $(t_p \in \{\mathsf{o}, \mathsf{mb}, \mathsf{rb}\}) \implies (B_2 = [(tag_q, t_q) \in B_0 \mid t_q = \mathsf{c}])$, $(t_p = c) \implies (B_2 = B_0)$

TABLE I: Effect of selected operations on borrow stack (S_B) in machine $\mathcal{M}_0$. Effects on R and M are not shown.

the type of p_1 depends on the type of p_0. The intent is for q_0 to have access rights till it surrenders them to p_1.

The copy operation creates two copies c_1 and c_2 using `cpy_mkcpy1` and `cpy_mkcpy2` instructions. A copied pointer corresponds to a raw pointer in Rust. The lender pointer p_0 for a copy operation can be of any *ptype*. Similar to a mutable borrow operation, all entries on top of p_0 are popped from the borrow stack and p_0 itself is removed. Next $c_1 :: c_2$ are pushed onto the borrow stack in that order. The *ptype* of c_1 is always c. However, the *ptype* of c_2 depends on the lender pointer p_0. This ensures that the *ptype* of a lender pointer is not lost through successive copy operations. Finally, the `die` operation surrenders access rights for a pointer by popping off its entry from the borrow stack. It is only defined for a mutably borrowed pointer q and signals transfer of data from such a pointer to its immediate lender, which must be of $ptype = \mathsf{o}$ or $ptype = \mathsf{mb}$. The pointer q must be top of borrow stack. The `die` operation is an extension of stacked borrows and is useful for returning information from a mutable borrow to the succeeding pointer without going through shared memory. The `store` instruction writes a value to memory. If the lender pointer p is mutably borrowed or owning then all elements before p are popped. If p is copied then borrow stack remains unchanged. The `load` instruction reads values from memory into a register using a lender pointer p. If p is owning, mutably borrowed or read-only borrowed, then all pointers above p (except copied pointers) are removed from the borrow stack. If p is copied, then the borrow stack is unchanged. Finally, the observable state $ObsState_{\mathcal{M}_0}$ of machine $\mathcal{M}_0$ is given by the tuple (P, R, M, S_B).

Let us look at an example of how $\mathcal{M}_0$ operates in Fig. 2. The intent of the program is to (1) create an owned pointer, (2) make its alias (3) update data through the alias, and, (4) observe the data through the owned pointer. At line 5, a word of memory is allocated with `(addr=0x4,tag=1)` in the register map at key `p0`, the integer `42` is written to memory at `M[0x4]`, and the *tag* value 1 is pushed to the borrow stack at `SB[0x4]`. Next an alias is created using the mutable borrow operation at lines 7–8 using tags `3` and `2` for borrowed `q0` and succeeding pointer `p1` respectively. First the tag for `p1` is pushed, then the tag for `q0` is pushed. The next couple of lines load `42` using `q0`, increment it, and write it back. The program ends the mutable borrow in line 14. This removes `q0`'s tag from `SB`. Now only `p1` can access *addr* `0x4`. Finally, the program reads the new value `43` from *addr* `0x4` in line 16.

Semantics of $\mathcal{M}_1$. We now define an extension to $\mathcal{M}_0$ called $\mathcal{M}_1$. In $\mathcal{M}_1$, a fat pointer additionally has a *cache* bit-vector field called *val*. Each store operation also updates *val* with the value to be written to memory. A load from memory may be replaced by *val* when correct to do so. A pointer value now becomes $(addr, tag, val)$. Overall, the semantics of existing instructions aim to maintain the *val* cache. The semantics is laid out in Table II. The `mk_own` instruction updates its cache with the value it initialized the memory allocation with. The pair of `mut_mkbor` and `mut_mksuc` operations have two cases: (1) if the lender is top-of-(borrow)stack then the operation reads the value stored at lender pointer p_0 and updates the caches of q_0 and p_1 with that value; (2) if the lender is not top of stack then the value at lender may be stale and the correct value is read from memory. The pair of `cpy_mkcpy1` and `cpy_mkcpy2`

Operation	Pre-condition	Post-condition
`p = mkown n`	–	$R[\mathsf{p}] = (p.addr, tag_p, n), M[p.addr] = n$

`q0 = mut_mkbor p0`
`p1 = mut_mksuc p0`
	$R[\mathsf{p_0}] = (p_0.addr, tag_{p_0}, v_p)$	$R[\mathsf{q_0}] = (p_0.addr, tag_{q_0}, v), R[\mathsf{p_1}] = (p_0.addr, tag_{p_1}, v)$,
		$(B_0 = \varnothing) \implies (v = v_p)$,
		$(B_0 \neq \varnothing) \implies (v = M[p_0.addr])$

`c1 = cpy_mkcpy1 p0`
`c2 = cpy_mkcpy2 p0`
	$R[\mathsf{p_0}] = (p_0.addr, tag_{p_0}, v_p)$	$R[\mathsf{c_1}] = (p_0.addr, tag_{c_1}, v), R[\mathsf{c_2}] = (p_0.addr, tag_{c_2}, v)$,
		$(B_0 = \varnothing \wedge t = \{\mathsf{o}, \mathsf{mb}, \mathsf{rb}\}) \implies (v = v_p)$,
		$\neg(B_0 = \varnothing \wedge t = \{\mathsf{o}, \mathsf{mb}, \mathsf{rb}\}) \implies (v = M[p_0.addr])$

`die q`	$R[\mathsf{q}] = (q.addr, tag_q, n)$	$R[p] = (q.addr, tag_p, n)$, $\exists p.R[p] = (q.addr, tag_p, \_)$
`m1 = store r, p, m0`	$R[\mathsf{p}] = (p.addr, tag_p, \_)$	$M[p.addr] = v, R[\mathsf{p}] = (p.addr, tag_p, v)$
`r = load p, m`	$R[\mathsf{p}] = (p.addr, tag_p, v_p)$	$R[\mathsf{r}] = v, R[\mathsf{p}] = (p.addr, tag_p, v)$, $((B_0 = \varnothing, t_p \in \{\mathsf{o}, \mathsf{mb}\}) \implies (v = v_p))$ $((B_0 \neq \varnothing \vee t_p = \mathsf{c}) \implies (v = M[p.addr]))$

TABLE II: Effect of selected operations on S_B,R, and M in machine $\mathcal{M}_1$ in addition to pre-and-post conditions from Table I.

```
fun main() {
BB0:
  m00 = mem.init()
  ;  R = [] | M = [] | SB = []
  p0,m0 = mk_own 42, m00
  ;  R[p0] = (0x4,1) | M[0x4] = 42 | SB[0x4] = 1 :: []
  q0 = mut_mkbor p0
  p1 = mut_mksuc p0
     R[p1] = (0x4,2) |   |
  ;
     R[q0] = (0x4,3) | M | SB[0x4] = 3 :: 2 :: []
  r1 = load q0, m0
   R[r1] = 42 | M | SB ;
  m1 = store r1 + 1,q0,m0
  ;  R | M[0x4] = 43 | SB
  die q0
  ;  R | M | SB[0x4] = 2 :: []
  r = load p1, m1
  ;  R[r] = 43 | M | SB
  halt
}
```

Fig. 2: Example of $\mathcal{M}_0$ operation. Effect on register map (R), memory map (M), and borrow store (S_B) shown in pink.

instructions similarly update the cache of c_1 and c_2 with the correct value. The `die` instruction transfers the value cached at q to the cache of the immediately succeeding pointer, called p here. The transfer to the succeeding pointer occurs by first searching for the pointer with the correct tag in the register map R and then updating the corresponding val field. Since we do not support the storage of pointers to memory, the search through R is enough to find the right pointer. Note that the `die` operation enables transfer of a value from a mutable borrow to the succeeding pointer without using shared memory. A `store` instruction updates the cache with the value `r` to be written to memory. This value is then written to memory and to $p.val$. In $\mathcal{M}_1$, a `store` does not support storing pointers to memory. This restriction is lifted in an extended version [11]. A `load` has two cases. First, if the lender pointer p is top-of-(borrow)stack, and is mutably borrowed or owning, then the read from memory is replaced by a read of the val (cache) field. Second, if the `load` uses a lender pointer p that is not top-of-(borrow)stack, or is copied, then the read from memory proceeds as usual. In the second case, the pointer cache is also updated with the value read from memory.

The optimisation we describe for the `load` instruction is correct because $\mathcal{M}_1$ always maintains the following invariant:

Theorem 1 (Cache equivalence). *For all pointers in the register map, if the pointer is top-of-(borrow)stack and is owning or mutably borrowed then the pointer cache value is the same as the value of memory at address of the pointer. Formally, let R be a register map, M memory, and S_B a borrow store. Then,*

$$
\begin{aligned}
&(R[p] = (addr, tag_p, n)) \wedge \\
&\qquad (S_B[addr] = (tag_p, t_p) :: B) \wedge \\
&\qquad\qquad (t_p \in \{\mathsf{o}, \mathsf{mb}\})) \implies M[addr] = n
\end{aligned}
$$

Proof. The proof proceeds by structural induction on the syntax of the program P. Assume Thm. 1 holds in some configuration (P_0, R_0, M_0, S_{B_0}). The next instruction takes the configuration to (P_1, R_1, M_1, S_{B_1}). We case-split on each possible instruction. We illustrate the process through some of the relevant instructions.

- `store` keeps the cache in-sync with memory according to given semantics;
- `mut_mkbor` keeps the mutably borrowed pointer cache in-sync with memory since the lender cache value is already in-sync (by assumption) and mutably borrowed pointer cache gets this value;
- `die`, before this `die` Thm. 1 holds for the mutably borrowed pointer. Then, `die` copies cache value from mutably borrowed pointer to succeeding pointer, keeping the succeeding pointer cache in-sync with memory. ■

We now define $ObsState_{M1}$ for $\mathcal{M}_1$ as a tuple (P, R, M, S) with the pointer val field excluded from view. Let $\equiv$ be the equivalence relation between $\mathcal{M}_0$ and $\mathcal{M}_1$ defined as follows: $s_{M0}^{m_0} \equiv s_{M1}^{m_1} \leftrightarrow ObsState_{M0}(s_{M0}) = ObsState_{M1}(s_{M1})$. By Thm. 1, starting in equivalent observable states, both $\mathcal{M}_0$ and $\mathcal{M}_1$ operate in lock-step. Thus, the following theorem holds:

Theorem 2. *The relation $\equiv$ is both a forward and a backward simulation between $\mathcal{M}_0$ and $\mathcal{M}_1$.*

Thus, safety of $\mathcal{M}_1$ implies safety of $\mathcal{M}_0$ and vise versa.

III. VC Generation

```
1  fun main() {
2  BB0:
3    m00 = mem.init()
4    $m_{00}$
5    p0,m0 = mk_own 42, m00
6    $p_0.addr = 4 \wedge p_0.val = 42 \wedge$
7    $m_0 = m_{00}[p_0.addr \mapsto 42]$
8    q0 = mut_mkbor p0
9    p1 = mut_mksuc p0
10   $q_0.addr = p_0.addr \wedge q_0.val = p_0.val \wedge q_0.retval = x \wedge$
11   $p_1.addr = p_0.addr \wedge p_1.val = x \wedge p_1.retval = p_0.retval$
12   r1 = load q0, m0
13   $r_1 = q_0.val$
14   m1 = store r1 + 1, q0, m0
15   $q_1.addr = q_0.addr \wedge q_1.retval = q_0.retval \wedge$
16   $q_1.val = r_1 + 1 \wedge m_1 = m_0[q_1.addr \mapsto q_1.val]$
17   die q0
18   $q_1.val = q_1.retval$
19   r = load p1, m1
20   $r = p_1.val$
21   assert r == 43
22   $\neg(r = 43)$
23   halt
24 }
```

Fig. 3: Verification condition (VC) shown in yellow.

We introduce the general encoding of an OSEA-IR program and the modelling of mutable borrows in particular using the example in Fig. 3. Note that this example runs throughout this section. For now, we suggest the reader ignore the generated VC (in yellow). We focus on aliasing instructions and how the pointer cache is affected. The `mk_own` instruction defines `p0` writing `42` to both memory and the pointer cache maintaining *Cache Equivalence*. The `mut_mkbor`, `mut_mksuc` instructions create aliases `q0`, `p1` from `p0`. Here, the cache at `p0` is copied to `q0` and `p1`, again maintaining the cache equivalence invariant. The `q0` mutably borrowed alias updates memory (and its pointer cache) to `43`. It then surrenders access rights using the `die` instruction. At this point, the succeeding alias `p1` becomes active (top-of-(borrow)stack). However, for `p1` to maintain cache equivalence (Theorem 1), it must get a copy of `q0`'s cache. This is not straightforward since there is no explicit transfer instruction from `q0` to `p1`. The standard solution is to use shared memory so that `q0` can write to this memory on a `die` and the succeeding pointer `p1` can then read from this memory on next access. However, the aim of caching is to eschew memory accesses as much as possible to keep the operation (and VC) simple. The concrete semantics of $\mathcal{M}_1$ provides one alternative to accessing memory. There, a `die` instruction finds the succeeding pointer tag in the borrow store S_B and then searches through the register map R to update the pointer cache with the same tag. This mechanism is as (or more) expensive to execute symbolically as shared memory. An elegant solution proposed in RustHorn [3] uses a *prophecy variable* [9] to model the return of a mutable borrow in the VC. We adapt the scheme to VC generation (VCGen) for OSEA-IR. We now explain VCGen, emphasizing the role of prophecy variables to model return of a mutable borrow.

The VC is generated using the sym translation function. It builds up the VC in a recursive, bottom-up fashion on the abstract syntax tree of an OSEA-IR program. For simplicity of presentation, we assume that two fundamental sorts are used in the encoding: bit-vector of 64 bits, $bv(64)$, and a map between bit-vectors, $bv(64) \to bv(64)$. We now revisit the example and explain the VC for each line of source code. Line 4 models `mem.init` as m_{00}, a free variable. Line 6 models the `mk_own` instruction. It updates memory at $m_{00}[addr]$ to 42 and defines the fat pointer p_0. A fat pointer is modelled as a tuple $(addr, val, retval)$. Here $addr$ holds the address, val holds the current cache value (42 here), and $retval$ holds a prophecy value, the use of which will be laid out soon. A mutable borrow operation occurs in lines 10–11. The lender pointer p_0 creates two aliases, the mutable borrow q_0 and the succeeding pointer p_1. The location $p_0.addr$ is copied to both $q_0.addr$ and $p_1.addr$. The cache at $p_0.val$ is copied to $q_0.val$. To set up the return of the cache value from the mutably borrowed alias to the succeeding pointer, we *entangle* the $q_0.retval$ and $p_1.val$ field using a fresh prophecy value x. This prophecy x will resolve to the correct cache value when `q0` dies. When this happens, p_1 instantly gets the same value in its cache in $p_1.val$. Moving ahead, lines 13–16 model the increment of the value pointed to by q_0. Note that apart from updating the value in memory, the $q_0.val$ variant $q_1.val$ also gets the updated value. Finally, in line 18, the `die` operation causes the prophecy x to be constrained by equating $q_1.val$ and $q_1.retval$. As expected, this defines $p_1.val$ to get the correct cache value 43 maintaining cache equivalence. The transfer of cache from `q0` to `p1` is, therefore, modelled without any expensive symbolic operations involving memory accesses or register map lookups. In the end, we see that the generated VC is unsatisfiable and the property is valid.

We now describe the function sym for selected pointer operations. The semantics of `mk_own` is given in Fig. 4. We assume that an address ℓ is given by an external allocator. The allocator should follow the usual property that ℓ has not been allocated previously. Note that $p_0.retval$ field is free since an owning pointer does not return the cache value to another alias. We define sym for mutable borrow and die operations in Fig. 5. The mutable borrow aliasing operation copies the $addr$ field from the lender to the borrower and succeeding pointer. The cache is wired as follows. First, the mutably borrowing pointer gets the lender cache using $q_0.val = p_0.val$. Second, we entangle $p_1.val$ with the free symbol $q_0.retval$ using the $tngle$ macro. The macro itself entangles the first argument with the second by equating them. Third, $p_1.retval$ gets the prophecy in $p_0.retval$ to model cascading borrows (reborrows). The sym for *die* equates the given pointer's val and $retval$ field, constraining the prophecy value in $q.retval$ and returning the borrow.

In summary, the fat pointer concept is our workhorse in mapping two previous high level VCGen schemes to a low-level verification setting. First, the reference elimination mechanism is replaced by fat pointers that cache values. Second, a fat pointer field holds a prophecy value that expresses the cache value after returning from a mutable borrow.

$$sym(\texttt{p0, m1 = mk\_own n, m0}) \triangleq \exists \ell.(m_1 = m_0[\ell \mapsto n]) \wedge (p_0.addr = \ell) \wedge (p_0.val = n)$$

Fig. 4: Definition of *sym* for mk_own.

$$tngle(r_1, r_2) \triangleq r_1 = r_2$$

$$sym(\texttt{q\_0 = mut\_mkbor p\_0; p\_1 = mut\_mksuc p\_0}) \triangleq q_0.addr = p_0.addr \wedge q_0.val = p_0.val \wedge tngle(p_1.val, q_0.retval) \wedge p_1.retval = p_0.retval$$

$$sym(\texttt{die q}) \triangleq q.val = q.retval$$

Fig. 5: Definition of *sym* for mutable borrow, die, and *tngle* macro for entanglement.

```
⟨RDEF⟩  ::=  ... | ⟨P⟩, ⟨M⟩ = mk_own ⟨R⟩, ⟨M⟩ |
             ⟨P⟩ = begin_unique ⟨P⟩ | ⟨P⟩ = end_unique ⟨P⟩ |
             ⟨P⟩ = set_cache ⟨P⟩, ⟨R⟩ | ⟨R⟩ = get_cache ⟨P⟩
```

Fig. 6: Grammar of new instructions for OSEA-IR.

```
1  extern void escapeToMemory(char *);
2  int main() {
3   char *p = MK_OWN(0, sizeof(char));
4   char c = nd_char();
5   assume (c == 42);
6   SET_CACHE(p, c);
7   *p = c;
8   char *b;
9   MUT_BORROW(b, p);
10  if (nd_bool()) {
11    c = nd_char();
12    assume(c > 43);
13    SET_CACHE(b, c);
14    *b = c;
15    escapeToMemory(b);
16  }
17  DIE(b);
18  char r;
19  GET_CACHE(p, r);
20  sassert(r == 42 || r > 43);
21  return 0;}
```

Fig. 7: A C program with Ownership macros in yellow.

IV. Towards a Practical Machine

In Section II, we described $\mathcal{M}_0$ and $\mathcal{M}_1$, both machines that could only allocate a single word through mk_own. We lift this restriction now in $\mathcal{M}_2$. To allocate multiple words (wide allocations), we change the mk_own syntax. Instead of taking a bit-vector to write to memory, it now takes a bit-vector *allocation size* argument. For cache equivalence to hold, the pointer cache width must now be wide enough to cache multi-byte allocation data. This complicates the design of the cache. To keep things simple, instead of hard-wiring the pointer cache to replicate memory contents, we only cache a *summary* of the data in memory and provide operations to set and get the cache value using set_cache and get_cache, respectively. A property to be verified can be cached at the pointer. Pointer aliasing operations copy the value as before. The decoupling of cache from load and store operations does introduce burden on the programmer to update the cache as required. As we move towards a practical machine, we also add a new unique (u) variant to pointer type *ptype*. A unique pointer is created using begin_unique and end_unique instructions.

The syntax of these new instructions is given in Fig. 6. The mk_own instruction takes three arguments - the bit-vector to write, the size (in bytes) of the allocation and the incoming memory to update. The operation now does not update memory or the pointer cache since that is the programmer's responsibility. The begin_unique and end_unique operations take a copied (unique) pointer and define a unique (copied) pointer with the same *addr* and *val* fields as the source pointer. These operations are useful when the user only wants to mark a pointer as unique temporarily. The get_cache instruction returns the *val* field of a pointer. the set_cache instruction takes a pointer and a value. It then defines a new pointer where all fields are the same as the source pointer, except the *val* field that has been updated to the given value.

Verification pipeline. To evaluate the efficacy of ownership intrinsics for verification, we use the SeaBMC bit-precise bounded model checker. SeaBMC operates on LLVM IR programs. For this work, the SeaBMC VCGen process has been enhanced to handle ownership instructions. It is cumbersome to construct low-level OSEA-IR programs by hand to be verified in SeaBMC. To ease the task, we provide an API for adding ownership semantics to C programs resulting in a C-like programming language with ownership semantics. The API is in the form of C macros. The C program is compiled to an OSEA-IR program. The low-level OSEA-IR program then generates the VC in SMT-LIB form. This is finally sent to an SMT solver. We discuss the API using the example high level program in Fig. 7. The program starts in line 3, the MK_OWN macro allocates a byte of memory to an owning pointer. The next line uses the nd_char function to assign a non-deterministic char to c. The value of c is constrained to be 42 using an **assume** statement. In line 6, the cache at pointer p is set to the value of c using the SET_CACHE macro. The value is also stored in memory using pointer p. The macro MUT_BORROW in line 9 then creates a mutable borrow. Internally, the macro expands to mut_mkbor and mut_mksuc with b getting the mutable borrow and p getting the succeeding pointer. Next, the non-deterministic boolean value from nd_bool is used in line 10 to conditionally update b's cache to a non-deterministic value greater than 43. The escapeToMemory function takes the address of b thwarting any optimisation attempts to promote b to a register. Finally, b dies in line 17 using the macro DIE. The succeeding pointer's cache is now read using GET_CACHE into r in line 19. The **sassert** (static assert) then checks that the value of r is either 42 or greater than 43.

For the program in Fig. 7, Fig. 8a is its OSEA-IR form and Fig. 8b is the generated VC. We now describe the VCGen in $\mathcal{M}_2$ using Fig. 8. The ownership instructions are highlighted in yellow in both figures. The MK_OWN macro in C becomes the mk_own instruction in OSEA-IR and is translated to SMT-LIB form using *sym*. Note that in $\mathcal{M}_2$, mk_own does not write to memory or update the pointer cache. The symbolic semantics therefore only allocates memory and provides a previously unallocated address $addr_0$. The set_cache instruction in line 7 defines a pointer p_3 with the same *addr* as p_2 and the cache updated to r_5. The mutable borrow occurs in lines 9–10. The

```
 1 fun main() {
 2 BB0:
 3   m3 = mem.init()
 4   p2, m0 = mk_own 1, m3
 5   r5 = nd_char()
 6   r6 = r5 == 42
 7   p3 = set_cache p2 r5
 8   m1 = store r5, p3, m0
 9   p5 = mut_mkbor p3
10   p6 = mut_mksuc p3
11   r15 = nd_bool();
12   r17 = r15 == 42
13   br r17, ERR, BB1
14
15 BB1:
16   r18 = nd_char()
17   r19 = r18 > 43
18   r20 = r6 && r19
19   p23 = set_cache p5 r18
20   m2 = store r18, p23, m1
21   escapeToMemory(p0)
22   br ERR
23
24 ERR:
25   r22 = select r17, r6, r20
26   p24 = select r17, p5, p23
27   die p24
28   r29 = get_cache p6
29   r30 = r29 == 42
30   r31 = r29 > 43
31   r32 = r30 || r31
32   A = not r32
33   assume A
34   assert false
35   halt
36 }
```

(a) OSEA-IR program.

$$
\begin{aligned}
&p_2.addr = addr_0 \land m_0 = m_3 \land \\
&r_6 = (r_5 = 0) \land \\
&p_3.addr = p_2.addr \land p_3.val = r_5 \land \\
&p_5.addr = p_3.addr \land p_6.addr = p_3.addr \land \\
&tngle(p_5.retval, p_6.val) \land p_5.val = p_3.val \land \\
&r_{17} = (r_{15} = 0) \land \\
&r_{19} = r_{18} > 1 \land \\
&r_{20} = r_6 \land r_{19} \land \\
&p_{23}.addr = p_5.addr \land p_{23}.val = r_{18} \land \\
&r_{22} = ite(r_{17}, r_6, r_{20}) \land \\
&p_{24} = ite(r_{17}, p_5, p_{23}) \land \\
&p_{24}.retval = p_{24}.val \land r_{22} \land \\
&r_{29} = p_6.val \land \\
&r_{30} = r_{29} = 0 \land \\
&r_{31} = r_{29} > 1 \land \\
&r_{32} = (r_{30} \lor r_{31}) \land \\
&a = \neg r_{32} \land \\
&a \land \\
&\neg false
\end{aligned}
$$

(b) SMT-LIB program.

Fig. 8: Program from Fig. 7 in OSEA-IR and SMT-LIB forms. Ownership intrinsics and their counterpart expressions in SMT are highlighted in yellow.

```
enum status {O, C};
int unit_proof(const char **fnames,
    int n) {
  FILE *f[MAX];// assume n < MAX
  for(int i=0; i < n; i++) {
    set_shad(f[i], O);
    f[i] = open(fnames[i], "w");}
  size_t choose = nd_size_t();
  assume(choose < n);
  FILE *file = f[choose];
  write(file);
  // check file closed
  sassert(get_shad(file) == C);}
```

(a) A unit proof.

```
void write(FILE *fp) {
  // check file opened
  sassert(get_shad(fp) == O);
  fputc('a', fp);
  // mark closed
  set_shad(fp, C);
  fclose(fp);}
```

(b) An SUT.

Fig. 9: An example of typestate storage in shadow memory.

semantics copies the lender $p_3.addr$ to $p_5.addr$ and $p_6.addr$. The val and $retval$ fields are set up as usual. The cache of the borrowed pointer p_5 is conditionally updated in line 19. The borrowed (variant) pointer p_{24} dies in line 27 with the usual semantics. The cache of the succeeding pointer p_6 is read into r_{29} in line 28. The lines 29–32 set up verification such that if an execution satisfies **assume** A then it reaches the error state (`assert false`). An important consequence of ownership semantics is that the SMT-LIB program does not need to model the `store` instruction in line 20.

V. Evaluation

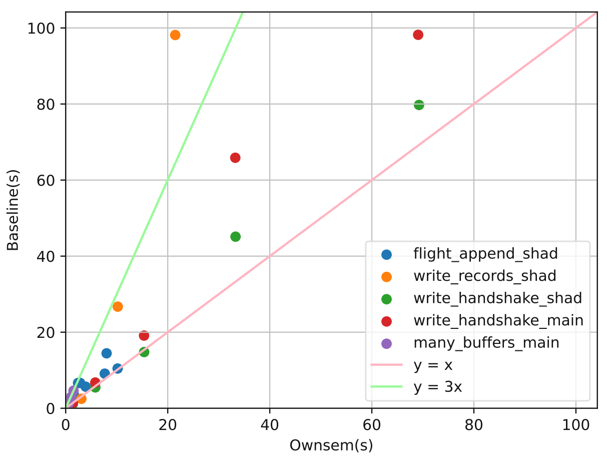


Fig. 10: Solve time (in sec.) using ownership semantics vs baseline.

We would like to cache verification properties for practical programs. We first describe properties of interest and our baseline property carrying mechanisms through the example in Fig. 9. Here, we want to check that a `write` function in Fig. 9b writes only to an *open* file and the file is always *closed* after a write. The state of the file object is encoded as a *typestate* property [13] that records (and checks) operations that have occurred on an object. The `write` function is called using the *unit proof* harness in Fig. 9a. It defines `n` file pointers based on user input. The typestate is marked open for each file pointer in *shadow memory* using the `set_shad` function. Shadow memory is an address map ($addr$ to $bv(64)$). It is used to stash verification relevant object metadata. In practice, shadow memory is provided by verification and testing tools like SeaBMC and Memcheck [14]. If shadow memory is not available then metadata can be stashed in a separate main (program) memory allocation, which is available when the program is built in debug or verification mode. In the example, after the typestate is set to open for each file pointer, we choose one file pointer `file` out of `n` for calling `write` — the system under test (SUT). In the SUT, we first check that the typestate of the file object is open using `get_shad` to get the typestate value. Then the char `'a'` is written and the file is closed, with the typestate marked as closed. Finally, the harness checks that the file typestate is indeed closed.

Note in the given unit proof, the write and read of shadow memory can resolve to `1`-of-`n` allocations since any file object can be chosen. Therefore, in the VC, memory access involves solving an ITE (if-then-else) expression for a choice. However, solving this ITE is redundant since we only want to check that a given operation occurred on the chosen file pointer. An alternative would be to store typestate in the file (fat) pointer cache itself. With this optimisation, an ITE would not be traversed since read, writes of the typestate would be at the pointer (using `get_cache` and `set_cache`) leading to simpler VC.

We base our experiments on this idea utilizing the SeaBMC model checking engine for SeaHorn. SeaBMC originally takes a SEA-IR program as input and generates VC that are solved by an SMT solver. We enhance SEABMC to now take OSEA-IR programs as input. Using the C macro API, C unit proofs are compiled to SMT. We then measure how typestate cached at pointers compares to typestate stored in memory.

The C unit proofs we work with come from `mbedTLS` [15], a C library of cryptographic primitives, SSL/TLS and DTLS protocols. In particular, we look at three functions in `ssl_msg.c` that handles SSL message construction and de-construction. The flow we consider are (1) `flight_append` that appends messages to the current flight of messages, (2) `write_records` that encrypts messages into records and sends them on the wire, and, (3) `write_handshake` that writes handshake messages. Each SUT operates on a byte buffer data structure. We are interested in recording and checking typestate properties for such a buffer. However, similar to example Fig. 9, the unit proof is set up such that a single byte buffer pointer may point to 1-of-n buffer objects. Therefore, we study if using pointer caching improves solver performance.

The experiments are run on an Intel(R) Xeon(R) E5-2680 CPU operating at 2.70GHz with 64 GiB of main memory. The generated VC are solved using Z3 [16] `smtfd` tactic. The scatter plot in Fig. 10 shows the solving time for unit proof with ownership semantics (ownsem) in the x-axis. The y-axis records the solving time for the same unit proof that either uses shadow memory or main memory as the baseline. The legend clarifies the memory we compare against using either *shad* or *main* in the name suffix. We run each flow for increasing number of byte buffers behind a pointer (e.g., 2, 4, 6, ...) and stop when the running time in either ownsem or baseline mode reaches 100 seconds. The `many_buffers` benchmark is hand-crafted and shows a consistent 3x improvement for ownsem. The flows from `mbedTLS` show more spread. For small number of buffers, ownsem and baseline are usually head-to-head. As the number of buffers increase, ownsem outperforms baseline. For `write_handshake_shad`, the performance boost is 1.3x when using 8 buffers. For `write_records_shad`, the performance boost at 8 buffers is 5x. This is shown on the scatter plot. Looking at the SMT solver metrics, we see internal metrics like `sat conflicts` and `sat backjumps` correlate with the timings (See [11] for details). When a unit proof is faster, fewer conflicts and backjumps are seen compared to the baseline. This is indirect evidence that the performance boost is due to VC simplicity.

Simplification of VC itself depends on how many memory accesses can be soundly replaced by pointer cache accesses. VC simplicity is affected by (1) the extent a conditional typestate check depends on program memory state, and (2) the number of typestate memory accesses as a fraction of the total number of memory accesses. As an example, for a conditional check such as `if(*unrelated_ptr == 1){get_cache(ptr);}`, a read of `ptr` cache using `get_cache` does not access `ptr` memory. However, for the check to be reachable, the guarding `if` condition does require a memory access. Therefore, it is not always possible to remove dependency on program memory for conditional typestate checks. Also if the unit proof (and the SUT) do not set/get typestate checks frequently then replacing such checks by pointer cache accesses has limited benefits. The data and units proofs to reproduce our experiments are available at https://github.com/priyasiddharth/mbedtls-ownsem.

Overall, a speedup in solving time occurs as expected. The speedup is due to simpler VC. However, the speedup is sensitive to the property expressed as a typestate check and number of operations on object (pointer) that affect typestate.

VI. RELATED WORK

RustBelt [17], Oxide [18] formalize subsets of high level Rust. RustBelt uses a continuation passing style functional language to describe the semantics. Oxide uses a high level language similar to Rust. These approaches do not map directly to a low–level register machine like LLVM. Stacked Borrows [12] formulates Rust ownership semantics as a stack discipline working on de–sugared (MIR) Rust syntax that represents memory by an address map. Its aim is to provide a reference semantics for the borrow checker separate from the production version in the Rust compiler. Stacked Borrows is implemented in the MIR interpreter (MIRI) and is part of the Rust standard distribution. We rely heavily on stacked borrows to design low-level semantics for this paper.

The Move Prover [5] uses reference elimination to replace a reference by its data. It assumes an alias free memory model and solves the problem of return of a mutable borrow by recording the origin (lender) of a mutable borrow and returning data to it explicitly rather than utilizing prophecies. RustHorn [3] uses a prophecy value to model return of a mutable borrow and assumes a safe Rust-like language and, therefore, forgoes modelling an address map entirely. RustHornBelt [6] extends this work to cover unsafe Rust where the safety in the unsafe part is manually proven in Iris [19], a concurrent separation logic prover built on top of Coq [20].

Verus [21], Prusti [22], and Creusot [23] are deductive verifers for Rust. Creusot uses RustHorn style prophecy variables. These deductive tools can model complicated features of the language, like polymorphism, directly. This paper focuses on low-level memory manipulating programs.

The memory models used in CBMC [24], LLBMC [25], and stock SeaBMC [10] assume an unsafe language allowing unrestricted aliasing of pointers and support pointer arithmetic. Kani [26] is a Rust verifier that compiles to goto-cc, the same low-level backend as CBMC. Ownership information, though, is lost is this conversion. Overall, we expect these low-level tools would perform similar to our baseline experiments.

VII. CONCLUSION

We describe formal ownership semantics for multiple low-level machines of increasing complexity. Particularly, we explain the mechanism for caching values at fat pointers and keeping the values in-sync with memory. We use the given semantics to describe VCGen for BMC such that the number of occurrences of memory accesses in the VC is reduced. For this we model return of mutable borrows using prophecy values added to fat pointers. We evaluate the efficiency of generated VC by experiments using the SEABMC tool. Overall, we see improvements in solving time and attribute it to the simplicity of VC.

References

[1] M. Fähndrich and R. DeLine, "Adoption and focus: Practical linear types for imperative programming," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 13–24. [Online]. Available: https://doi.org/10.1145/512529.512532

[2] D. Grossman, J. G. Morrisett, T. Jim, M. W. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 282–293. [Online]. Available: https://doi.org/10.1145/512529.512563

[3] Y. Matsushita, T. Tsukada, and N. Kobayashi, "Rusthorn: Chc-based verification for rust programs," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 4, pp. 15:1–15:54, 2021. [Online]. Available: https://doi.org/10.1145/3462205

[4] J. Noble, J. Vitek, and J. Potter, "Flexible alias protection," in *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, ser. Lecture Notes in Computer Science, E. Jul, Ed., vol. 1445. Springer, 1998, pp. 158–185. [Online]. Available: https://doi.org/10.1007/BFb0054091

[5] D. L. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and J. E. Zhong, "Fast and reliable formal verification of smart contracts with the move prover," in *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*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 183–200. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_10

[6] Y. Matsushita, X. Denis, J. Jourdan, and D. Dreyer, "Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 841–856. [Online]. Available: https://doi.org/10.1145/3519939.3523704

[7] L. Developers. (2024) llvm website. [Online]. Available: https://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata

[8] R. Developers. (2024) Rust website. [Online]. Available: https://github.com/rust-lang/rust/issues/54878#issuecomment-429578187

[9] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90224-P

[10] S. Priya, Y. Su, Y. Bao, X. Zhou, Y. Vizel, and A. Gurfinkel, "Bounded model checking for LLVM," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 214–224. [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_28

[11] S. Priya and A. Gurfinkel, "Ownership in low-level intermediate representation," 2024. [Online]. Available: https://arxiv.org/abs/2408.04043

[12] R. Jung, H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for rust," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 41:1–41:32, 2020. [Online]. Available: https://doi.org/10.1145/3371109

[13] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, 1986.

[14] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*, C. Krintz, S. Hand, and D. Tarditi, Eds. ACM, 2007, pp. 65–74. [Online]. Available: https://doi.org/10.1145/1254810.1254820

[15] mbedTLS Developers. (2023) mbedtls project. [Online]. Available: https://github.com/Mbed-TLS/mbedtls

[16] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[17] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, 2018. [Online]. Available: https://doi.org/10.1145/3158154

[18] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, "Oxide: The essence of rust," *CoRR*, vol. abs/1903.00982, 2019. [Online]. Available: http://arxiv.org/abs/1903.00982

[19] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds. ACM, 2015, pp. 637–650. [Online]. Available: https://doi.org/10.1145/2676726.2676980

[20] C. Developers. (2024) The coq proof assistant. [Online]. Available: https://coq.inria.fr/

[21] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 286–315, 2023. [Online]. Available: https://doi.org/10.1145/3586037

[22] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260. Springer, 2022, pp. 88–108. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_5

[23] X. Denis, J. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of rust programs," in *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, A. Riesco and M. Zhang, Eds., vol. 13478. Springer, 2022, pp. 90–105. [Online]. Available: https://doi.org/10.1007/978-3-031-17244-1_6

[24] D. Kroening, P. Schrammel, and M. Tautschnig, "CBMC: the C bounded model checker," *CoRR*, vol. abs/2302.02384, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.02384

[25] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of C and C++ programs using a compiler IR," in *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Joshi, P. Müller, and A. Podelski, Eds., vol. 7152. Springer, 2012, pp. 146–161.

[26] K. Developers. (2023) The kani book. [Online]. Available: https://model-checking.github.io/kani/

The Conference on Formal Methods in Computer-Aided Design (FMCAD) is an annual conference on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.

ISBN 978-3-85448-065-5

9 783854 480655

www.tuwien.at/academicpress