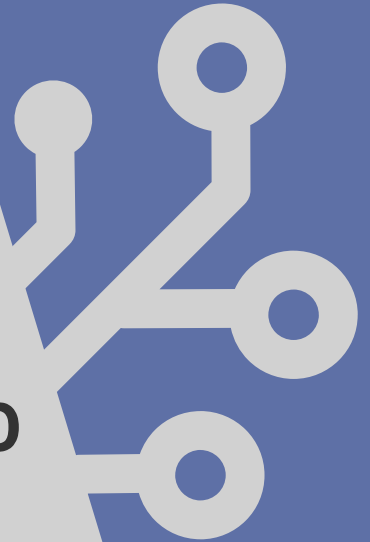


Ahmed Irfan / Daniela Kaufmann (Eds.)

PROCEEDINGS OF THE 25TH CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2025



Academic Press



fmcad.²⁵

Ahmed Irfan / Daniela Kaufmann (Eds.)

PROCEEDINGS OF THE 25TH CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED
DESIGN – FMCAD 2025

Conference Series: Formal Methods in Computer-Aided Design

Volume 6

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 6 edited by:

Ahmed Irfan, SRI, 333 Ravenswood Ave, Menlo Park, CA 94025 USA | ahmed.irfan@sri.com
Daniela Kaufmann, TU Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria | daniela.kaufmann@tuwien.ac.at

Ahmed Irfan / Daniela Kaufmann (Eds.)

PROCEEDINGS OF THE 25TH CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2025

Cite as:

Ahmed, I., & Kaufmann, D. (Eds.). (2025). *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design – FMCAD 2025*. TU Wien Academic Press. <https://doi.org/10.34727/2025/isbn.978-3-85448-084-6>

TU Wien Academic Press, 2025

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



This work is licensed under a Creative Commons attribution 4.0 international license (CC BY 4.0).
<https://creativecommons.org/licenses/by/4.0/>

ISBN (online): 978-3-85448-084-6
ISSN (online): 2708-7824

Available online: <https://doi.org/10.34727/2025/isbn.978-3-85448-084-6>

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): Ahmed Irfan and Daniela Kaufmann

Preface

These are the proceedings of the twenty-fifth International Conference on Formal Methods in Computer-Aided Design (FMCAD), held in Menlo Park, California, USA, from October 6–10, 2025. The first FMCAD was organized in 1996, and the conference was held biennially until 2006, when it merged with the CHARME conference to become a single annual event. Since then, FMCAD has been held every year. FMCAD 2025 marked the twenty-fifth edition in the series, covering formal aspects of computer-aided system design, including verification, specification, synthesis, and testing. It served as a leading forum for researchers from academia and industry to present and discuss groundbreaking methods, technologies, theoretical results, and tools for formal reasoning about computing systems. The FMCAD 2025 program included two tutorials, two invited talks, the presentation of the Hardware Model Checking Competition (HWMCC’25), a student forum, and the main program featuring presentations of 27 peer-reviewed, accepted papers.

FMCAD 2025 was co-located with the VSTTE 2025 conference, which took place on October 6–7.

The joint VSTTE/FMCAD tutorial day (October 7) featured three tutorials:

- The VSTTE tutorial: *EasyCrypt*, by Pierre-Yves Strub
- The FMCAD tutorials:
 - *Verification Modulo Theories*, by Alberto Griggio
 - *Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods*, by Ankush Desai

The main FMCAD conference (October 8–10) featured two invited talks:

- *Program Synthesis: Pre-LLM and Post-LLM* by Ashish Tiwari
- *Integrating Large Language Models in Automated Program Verification* by Nina Narodytska

FMCAD 2025 received 82 abstracts, which resulted in 64 full submissions. Of these, the program committee selected 27 papers for publication. Each submission underwent a rigorous review process, receiving at least four reviews. The topics of the accepted papers span hardware and software validation, model checking, machine learning, SAT and SMT solving, and proof generation. Among the accepted papers, 17 are regular papers (15 long and 2 short), and 10 are tool or case study papers (6 long and 4 short). This year, FMCAD introduced voluntary artifact evaluation for the first time. The artifacts were assessed by selected members of the program committee. A total of 20 submissions included artifacts, 12 of which were associated with accepted papers.

FMCAD 2025 hosted the thirteenth edition of the FMCAD Student Forum, which has been held annually since 2013. The forum provides a platform for graduate students at any stage of their academic career to present their research to the FMCAD community. The FMCAD Student Forum 2025 was organized by Tanja Schindler and Lee Barnett and featured short presentations of 20 accepted contributions. The proceedings include a detailed description of the Student Forum and list all accepted contributions.

FMCAD 2025 was made possible through the support of many individuals and our generous sponsors. The program committee members and additional reviewers, listed on the following pages, provided detailed and insightful reviews. Their efforts not only helped us assemble a strong technical program but also guided authors in improving their submissions. We sincerely thank each and every one of them for dedicating their time and expertise.

We would like to thank the local organization chair, Stéphane Graham-Lengrand, and the registration chairs, Jenny McNeill and Trish Carrillo, who expertly managed the logistics and practical aspects of the conference. We thank our web master Thomas Hader, our sponsorship chair Alex Ozdemir, and the Student Forum organizers Tanja Schindler and Lee Barnett. We also thank the organizers of the HWMCC competition, Armin Biere, Nils Froleys, and Mathias Preiner. Special thanks go to Georg Weissenbacher for his exceptional assistance in organizing the event, for serving as a liaison with the steering committee, and for his role as publication chair.

A conference like FMCAD would not be possible without the support of our sponsors. We gratefully acknowledge the contributions of (listed in alphabetical order): AWS, Cadence Design Systems Inc., Futurewei, General Electric Aerospace, Siemens, SRI, and TU Wien.

Last but not least, we thank all the authors who submitted their work to FMCAD 2025. Their contributions and presentations form the heart of the conference.

The conference proceedings are published as Open Access by TU Wien Academic Press, and are also available 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 2025

Daniela Kaufmann	TU Wien, Austria
Ahmed Irfan	SRI, USA

Organizing Committee

Program Co-Chairs

Ahmed Irfan	SRI, USA
Daniela Kaufmann	TU Wien, Austria

Local Organization Chair

Stéphane Graham-Lengrand	SRI, USA
--------------------------	----------

Registration Chairs

Jenny McNeill	SRI, USA
Trish Carrillo	SRI, USA

Student Forum Chairs

Tanja Schindler	University of Basel, Switzerland
Lee A. Barnett	Amazon Web Services, USA

Sponsorship Chair

Alex Ozdemir	Stanford University, USA
--------------	--------------------------

Web Chair

Thomas Hader	TU Wien, Austria
--------------	------------------

Publication Chair

Georg Weissenbacher	TU Wien, Austria
---------------------	------------------

FMCAD Steering Committee

Clark Barrett	Stanford University, CA, USA
Armin Biere	University of Freiburg, Germany
Ruzica Piskac	Yale University, CT, USA
Anna Slobodova	Arm, TX, USA
Georg Weissenbacher	TU Wien, Austria

Board of the FMCAD Association

Armin Biere	University of Freiburg, Germany
Roderick Bloem	Graz University of Technology, Austria
Georg Weissenbacher	TU Wien, Austria
Florian Zuleger	TU Wien, Austria

Program Committee

FMCAD 2025 Program Committee

Ahmed Irfan (co-chair)	SRI International
Daniela Kaufmann (co-chair)	TU Wien
Erika Ábrahám	RWTH Aachen University
Guy Amir	Cornell University
Kshitij Bansal	Google
Haniel Barbosa	Universidade Federal de Minas Gerais
Per Bjesse	Synopsys Inc.
Nikolaj Bjørner	Microsoft
Martin Blicha	University of Lugano
Roderick Bloem	Graz University of Technology
Aleksandar Chakarov	Phase Change Software LLC
Supratik Chakraborty	IIT Bombay
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Katalin Fazekas	TU Wien
Pascal Fontaine	Université de Liège, Belgium
Divya Gopinath	NASA Ames (KBR Inc.)
Alberto Griggio	Fondazione Bruno Kessler
Arie Gurfinkel	University of Waterloo
Liana Hadarean	Amazon Web Services
Osman Hasan	National University of Sciences and Technology (NUST)
Paula Herber	University of Münster
Marijn Heule	Carnegie Mellon University
Antti Hyvärinen	Certora
Alexey Ignatiev	Monash University
Mitesh Jain	Northeastern University
Mikoláš Janota	Czech Technical University in Prague
Susmit Jha	SRI International
Martin Jonáš	Masaryk University, Czechia
Jianwen Li	East China Normal University
Enrico Magnago	Amazon Web Services
Sergio Mover	Ecole Polytechnique
Antonina Nepeivoda	Program System Institute of RAS
Aina Niemetz	Stanford University
Mathias Preiner	Stanford University
Stefan Ratschan	Institute of Computer Science, Czech Academy of Sciences
Kristin Yvonne Rozier	Iowa State University
Philipp Rümmer	University of Regensburg
Mark Santolucito	Barnard College
Christoph Scholl	University of Freiburg
Martina Seidl	Johannes Kepler University Linz
Natarajan Shankar	SRI International
Natasha Sharygina	University of Lugano, Switzerland

Anna Slobodova	Arm
Mate Soos	Ethereum Foundation
Christoph Stickse	The MathWorks
Ashish Tiwari	Microsoft
Nestan Tsiskaridze	Stanford University
Georg Weissenbacher	TU Wien
Haoze Wu	Amherst College
Nisansala Yatapanage	Australian National University
Cunxi Yu	University of Maryland, College Park
Emily Yu	Institute of Science and Technology Austria
Hongce Zhang	Hong Kong University of Science and Technology (Guangzhou)
Zhen Zhang	Utah State University
Yoni Zohar	Bar-Ilan University

FMCAD 2025 Student Forum Committee

Tanja Schindler (co-chair)	University of Basel
Lee Barnett (co-chair)	Amazon Web Services
Armin Biere	University of Freiburg
Roderick Bloem	Graz University of Technology
Julie Cailler	University of Lorraine, CNRS, Inria, LORIA, Nancy, France
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Deepak D'Souza	Indian Institute of Science
Constantin Enea	Ecole Polytechnique
Mathias Fleury	University of Freiburg
Arie Gurfinkel	University of Waterloo
Clemens Hofstadler	Johannes Kepler University Linz
Petra Hozzová	Czech Technical University
Marie-Christine Jakobs	Ludwig-Maximilians-Universität München
Tim King	AWS
Katherine Kosaian	University of Iowa
Kasper Luckow	Amazon Web Services
Jan Strejček	Masaryk University
Jiyuan Wang	University of California, Los Angeles
Emily Yu	Institute of Science and Technology Austria

Additional Reviewers

Ashraf, Sobia
Aurandt, Alexis

Ciesielski, Maciej
Cobb, Adam

Davis, Mason
Dutta, Souradeep

Elderhalli, Yasmeen

Fleury, Mathias

Jacks Jr, Michael

Karimi, Mahyar
Kauers, Manuel
Kaur, Ramneet
Kolárik, Tomáš
Konrad, Alexander
Kovács, József
Kumar, Ankit

Leopardi, Fabrizio
Liang, Chencheng
Lu, Zhengyang

Maderbacher, Benedikt
Mascarenhas, Tomaz

Nukala, Karthik

Otoni, Rodrigo

Priya, Siddharth

Rao, Vikas
Reichl, Franz-Xaver
Ribeiro, Caio

Saidi, Hassen
Sarwar, Muhammad Bilal
Seufert, Tobias
Soldevila, Mallku
Su, Yusen
Swords, Sol

Tafese, Joseph

Zaman, Eshita

Table of Contents

Tutorials

Verification Modulo Theories	1
<i>Alberto Griggio</i>	
Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods	2
<i>Ankush Desai</i>	

Invited Talks

Program Synthesis: Pre-LLM and Post-LLM	3
<i>Ashish Tiwari</i>	
Integrating Large Language Models in Automated Program Verification	4
<i>Nina Narodytska</i>	

Student Forum

The FMCAD 2025 Student Forum	5
<i>Tanja Schindler and Lee Barnett</i>	

Hardware Model Checking Competition

Hardware Model Checking Competition 2025	7
<i>Armin Biere, Nils Froleys and Mathias Preiner</i>	

Temporal Logic

“How Does my Circuit Work?”: Local Explanations for the Behavior of Sequential Circuits	8
<i>Amirmohammad Nazari, Matin Amini, and Mukund Raghothaman</i>	
On Hyperproperty Verification, Quantifier Alternations, and Games under Partial Information	19
<i>Raven Beutner and Bernd Finkbeiner</i>	
Scalable MLTL Runtime Monitoring and Satisfiability via Bit-Vector Encoding	30
<i>Christopher Johannsen, Phillip H. Jones, Kristin Yvonne Rozier, and Tichakorn Wongpiromsarn</i>	

Neural Networks and Large Language Models

PolyVer: A Compositional Approach for Polyglot System Modeling and Verification	41
<i>Pei-Wei Chen, Shaokai Lin, Adwait Godbole, Ramneet Singh, Elizabeth Polgreen, Edward A. Lee, Sanjit A. Seshia</i>	
Quantifying Robustness of Medical Image Segmentation Networks Using TensorStars	54
<i>Meghan Stuart and Parasara Sridhar Duggirala</i>	

Of Good Demons and Bad Angels: Guaranteeing Safe Control under Finite Precision	65
<i>Samuel Teuber, Debasmitha Lohar, and Bernhard Beckert</i>	
Can Large Language Models Autoformalize Kinematics?	78
<i>Aditi Kabra, Jonathan Laurent, Sagar Bharadwaj, Ruben Martins, Stefan Mitsch, and André Platzer</i>	
SAT and SMT	
Towards SMT Solver Stability via Input Normalization	84
<i>Daneshvar Amrollahi, Mathias Preiner, Aina Niemetz, Andrew Reynolds, Moses Charikar, Cesare Tinelli, and Clark Barrett</i>	
Per-Instance Subproblem Generation for Strategy Selection in SMT	94
<i>Amalee Wilson, Nina Narodytska, Clark Barrett, and Haoze Wu</i>	
Solving Set Constraints with Comprehensions and Bounded Quantifiers	104
<i>Mudathir Mohamed, Nick Feng, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Marsha Chechik</i>	
Learning Short Clauses via Conditional Autarkies	115
<i>Amar Shah, Twain Byrnes, Joseph Reeves, and Marijn J. H. Heule</i>	
Tools	
R2U2 Playground: Visualization of a Real-time, Temporal Logic Runtime Monitor	126
<i>Alexis Aurandt, Kristin Yvonne Rozier, and Phillip H. Jones</i>	
s2s: An Eager SMT Solver for Strings	133
<i>Kevin Lotz, Mitja Kulczynski, and Dirk Nowotka</i>	
FastPoly: An Efficient Polynomial Package for the Verification of Integer Arithmetic Circuits	139
<i>Alexander Konrad and Christoph Scholl</i>	
OSTRICH2: Solver for Complex String Constraints	145
<i>Matthew Hague, Denghang Hu, Artur Jeż, Anthony W. Lin, Oliver Markgraf, Philipp Rümmer, Zhilin Wu</i>	
Case Studies	
A Formal Y86 Simulator with CHERI Features	159
<i>Carl Kwan, Yutong Xin, and William D. Young</i>	
A Method for the Verification of Memory Management Software in the Presence of TLBs	169
<i>Yahya Sohail and Warren A. Hunt, Jr.</i>	
Verification Application	
Making Rabbit Run for Security Verification of Networked Systems with Unbounded Loops	178
<i>Sewon Park and Atsushi Igarashi</i>	
Modeling the AWS Authorization Engine	188
<i>Lee A. Barnett, Loris D'Antoni, Amit Goel, Rami Gökhan Kkıcı, Neha Rungta, Mary Southern, and Chungha Sung</i>	

Automated Translation Validation of a Compiler for Statically Scheduled Accelerators	198
<i>Jackson Melchert, Caleb Terrill, Aron Ricardo Perez-Lopez, Clark Barrett, and Priyanka Raina</i>	
Unifying DQMax#SAT and DSSAT: Polynomial-Time Reduction and Applications	209
<i>Ilo Chen, Che Cheng, and Jie-Hong Roland Jiang</i>	

Synthesis

Synthesiz3 This: an SMT-Based Approach for Synthesis with Uncomputable Symbols	215
<i>Petra Hozzová, Nikolaj Bjørner</i>	
Guiding Likely Invariant Synthesis on Distributed Systems with Large Language Models.....	226
<i>Yuan Xia, Aabha Shailesh Pingle, Deepayan Sur, Srivatsan Ravi, Mukund Raghothaman, and Jyotirmoy V. Deshmukh</i>	
Unlocking Hardware Verification with Oracle Guided Synthesis	235
<i>Leiqi Ye, Yixuan Li, Guy Frankel, Jianyi Cheng, and Elizabeth Polgreen</i>	

Software Verification

Automated Formal Verification of a Software Fault Isolation System.....	246
<i>Matthew Sotoudeh and Zachary Yedidia</i>	
Static Coverage in Deductive Software Verification	251
<i>Aaron Tomb and Anjali Joshi</i>	
A Tale of Two Case Studies: A Unified Exploration of Rust Verification with SEABMC	262
<i>Joseph Tafese, Siddharth Priya, Giuliano Losa, Arie Gurfinkel, and Graydon Hoare</i>	

Verification Modulo Theories

Alberto Griggio 
Fondazione Bruno Kessler
Trento, Italy
griggio@fbk.eu

Abstract—In this tutorial, I will present techniques for automated formal verification of a variety of systems using Satisfiability Modulo Theories (SMT) solvers as main reasoning engines. The talk will focus mainly on the problem of verifying invariant properties on symbolic transition systems, but it will also cover techniques for the verification of Linear Time temporal logic (LTL) properties and extensions to deal with parameterized systems with an unbounded number of components. I will also introduce some concrete tools to solve Verification Modulo Theories problems automatically, and present a demo of their use in a realistic application.

Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods

Ankush Desai 

Amazon Web Services

San Jose, CA

ankushdesai@gmail.com

Abstract—Building reliable and secure software requires a range of approaches to reason about systems correctness. Alongside industry-standard testing methods (such as unit and integration testing), AWS has adopted model checking, fuzzing, property-based testing, fault-injection testing, deterministic simulation, event-based simulation, and runtime validation of execution traces. Formal methods have been an important part of the development process – perhaps most importantly, formal specifications as oracles or monitors that provide the correct answers for many of AWS’s testing practices. Correctness testing and formal methods remain key areas of investment at AWS. In this talk, we will cover some of these tools and techniques that have had impact within AWS and then also open challenges that remains to be solved to further amplify the adoption of formal methods in practice.

Program Synthesis: Pre-LLM and Post-LLM

Ashish Tiwari 

Microsoft Research

Redmond, WA

ashish.tiwari@microsoft.com

Abstract—This talk will cover the exciting progress made in the field of Program Synthesis over the years. Program synthesis is broadly concerned with generating programs from specifications. Program synthesis is a fascinating field since it encompasses a whole spectrum of problems that are obtained by just considering different notions of specifications and different classes of programs. We will discuss general approaches that have been developed, including bottom-up synthesis, top-down synthesis, and constraint-based synthesis. We will also cover some applications of program synthesis. The advent of large language models had an influence on the field, and in the second part of the talk, we will cover more recent work on program synthesis in this new age of large language models, along with the new opportunities it presents.

Integrating Large Language Models in Automated Program Verification

Nina Narodytska 

VMware Research by Broadcom

USA

n.narodytska@gmail.com

Abstract—The demonstrated code-understanding capabilities of large language models (LLMs) raise the question of whether they can be used for automated program verification—a task that typically demands high-level, abstract reasoning about program properties, which remains challenging for existing verification tools. In this talk, we overview several research directions, developed over the past two years, that propose methodologies for combining the strengths of LLMs and automated reasoners in program verification. We then take a closer look at Lemur, a tool that formally describes such a methodology as a set of derivation rules and proves its soundness. Lemur instantiates the calculus as a sound automated verification procedure and demonstrates practical improvements on a suite of synthetic and competition benchmarks.

The FMCAD 2025 Student Forum

Tanja Schindler 
 University of Basel
 Basel, Switzerland
 tanja.schindler@unibas.ch

Lee A. Barnett 
 Amazon Web Services
 Portland, Oregon, USA
 lbarn@amazon.com

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 2025, the event took place in Menlo Park, California, USA. Twenty 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 2025 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]
- Prague, Czech Republic in 2024 [12]

FMCAD 2025 hosted the thirteenth 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, of which 22 were reviewed and 20 were accepted. The Student Forum program committee reviews were based on the overall quality, novelty of the work, its potential impact on the formal methods community, and 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 systems. Each submission received 3 reviews. The following contributions have been accepted¹:

- Nikil Shyamsunder: *Work-In-Progress: Building an Interpreter for an Imperative Hardware Interface Specification Language*
- Mason Davis: *VeRAPak: Verification of Neural Network Robustness Through Refutation and Attack Methods*
- Joseph Tafese: *Actionable Feedback for eBPF Developers*

¹Only first authors listed for brevity.

- Era Thaqi: *Symbolic Timing Analysis of Digital Circuits Using Analytic Delay Functions*
- Pedro Saccomani: *Towards a Proof-Producing Theory of Finite Fields*
- Yao Hsiao: *Consistency-Directed Formal Verification of Cache Coherence Protocol Implementations*
- Chuyue Sun: *Automated Verification of Data-Structure Modules in Verus*
- Liam Davis: *Lookahead Branching for Neural Network Verification*
- Sergei Leonov: *Two Strategies to Improve the Stålmarck Procedure*
- Joshua Jeppson: *Reasoning About Quantitative Rare-Event Reachability in Stochastic Vector Addition Systems via Affine Vector Spaces*
- Ian Dardik: *Towards Fully Automated Compositional Inductive Invariant Inference*
- Landon Taylor: *Tackling Scalability for Transient Reachability Analysis of Chemical Reaction Networks*
- Anna Eaton: *SimSpect: Automated Litmus Testing for Simulated Spectre Defenses*
- William Fishell: *Synthesizing State Space Models: Learning Meets Logic*
- Bingqing Hu: *Error-Guided Predicate Abstraction for Continuous-Time Markov Chains*
- Jibiana Jakpor: *Verifying cvc5's String Rewrite Rules Using Isabelle/HOL*
- Alexis Aurandt: *Towards Verified Runtime Monitors for Resource-Constrained Embedded Systems*
- Sam Akinwande: *An Inductive Approach to Verifying Neural Feedback Systems*
- Nick Waddoups: *Verification of Digital Network-on-Chip Systems Using Dafny*
- Thomas Hader: *Optimization for MCSat*

We formed a program committee to cover a wide range of topics so that students could receive expert feedback on their work. The 2025 FMCAD Student Forum program committee consisted of Tanja Schindler (co-chair), Lee A. Barnett (co-chair), Armin Biere, Nikolaj Bjørner, Roderick Bloem, Julie Cailler, Deepak D'Souza, Rayna Dimitrova, Constantin Enea, Mathias Fleury, Arie Gurfinkel, Clemens Hofstadler, Petra Hozzová, Marie-Christine Jakobs, Tim King, Katherine Kosaiian, Kasper Luckow, Mark Santolucito, Jan Strejček, Jiyuan Wang, and Emily Yu.

We would like to thank the organizers of FMCAD, as well as the FMCAD Student Forum program committee, who made the FMCAD Student Forum possible. We would also like to thank the U.S. National Science Foundation, FMCAD, Amazon Web Services, Cadence Design Systems, Futurewei, General Electric Aerospace, Siemens, SRI International, and TU Wien, for providing student travel support. In addition, 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, p. 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, p. 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, p. 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, p. 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, p. 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, p. 1, https://fmcad.org/FMCAD21/student_forum/.
- [10] M. Preiner, “The FMCAD 2022 student forum,” in *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2022, pp. 5–6, 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. 1–2, https://fmcad.org/FMCAD23/student_forum/.
- [12] M. Blicha and N. Tsiskaridze, “The FMCAD 2024 student forum,” in *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2024, pp. 5–6, https://fmcad.org/FMCAD24/student_forum/.

Hardware Model Checking Competition 2025

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

Nils Froleys 
Johannes Kepler University
Linz, Austria
nils.froleys@jku.at

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

Abstract—The Hardware Model Checking Competition 2025 (HWMCC’25) was the 13th competitive event for hardware model checkers. Affiliated to the 25th conference on Formal Methods in Computer-Aided Design 2025 (FMCAD’25) it took place in Menlo Park, California, USA, from Oct. 6 to 10, 2025.

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

The *Hardware Model Checking Competition* in its 2025 edition (HWMCC’25) is the 13th incarnation in this series of competitive events to evaluate hardware model checking. Since it started in 2007 it was repeated annually with few exceptions. After the competition in 2020 the organizers took a break to resume the competition in 2024. The competition in 2025 was 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. In 2007, 2008, 2010 and 2014 it was affiliated with the conference of *Computer-Aided Verification* (CAV).

Since 2019, HWMCC features two *word-level* tracks, which focus on bit-vector models, one track *with arrays* and one *without arrays*, both in the BTOR2 format [1]. Prior to 2019, all competition tracks used *bit-level* models in the AIGER format [2], which were split into *safety*, *multi-property*, *liveness* and *deep* tracks. In 2024, participating model checkers submitted to the bit-level safety track were required to produce model-checking *certificates* [3]. These certificates are AIGER circuits that should have an inductive property and need to simulate the original circuit as formalized in [4], [5], [6]. The tool CERTIFAIGER was 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].

The introduction of certificates for the bit-level safety track of HWMCC’24 was a big success, with eight competing model checkers producing certificates. The model checkers were able to produce compact and correct certificates that were verified with minimal overhead. In HWMCC’24 the certifying winner of the bit-level safety track rIC3 [8] was even able to outperform the previous non-certifying state-of-the-art model checker ABC [9]. The results of mandatory certificates in HWMCC’24 demonstrated that certification can be adopted without compromising model checking efficiency [3]. The HWMCC’24 case-study [3] became a distinguished paper.

In 2025, the competition (HWMCC’25) continued with the word-level and bit-level tracks, but reintroduced the bit-level liveness track. In this track, participating model checkers were required to produce certificates for satisfiable liveness properties. Certificates for unsatisfiable liveness properties were optional, but may be required in future editions of HWMCC.

More details on the competition, including provided tools, submission procedure and deadlines, results and their presentation are available at <https://hwmcc.github.io/2025>.

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 1*, 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] N. Froleys, E. Yu, M. Preiner, A. Biere, and K. Heljanko, “Introducing certificates to the hardware model checking competition,” in *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part 1*, ser. Lecture Notes in Computer Science, R. Piskac and Z. Rakamaric, Eds., vol. 15931. Springer, 2025, pp. 281–295. [Online]. Available: https://doi.org/10.1007/978-3-031-98668-0_14
- [4] E. Yu, N. Froleys, 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. Froleys, 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 1*, ser. Lecture Notes in Computer Science, C. Benz Müller, 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] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang, “The ric3 hardware model checker,” in *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part 1*, ser. Lecture Notes in Computer Science, R. Piskac and Z. Rakamaric, Eds., vol. 15931. Springer, 2025, pp. 185–199. [Online]. Available: https://doi.org/10.1007/978-3-031-98668-0_9
- [9] 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.

“How Does my Circuit Work?”: Local Explanations for the Behavior of Sequential Circuits

Amirmohammad Nazari*, Matin Amini* and Mukund Raghothaman*

*University of Southern California

Los Angeles, CA, USA

{nazaria, matinami, raghotha}@usc.edu

Abstract—There has been a massive amount of work on algorithms to verify and synthesize systems from temporal specifications. In contrast, there has been less work devoted to the problem of helping engineers to understand how and why their systems exhibit certain behaviors. Such understanding is important for them to debug, validate, and modify their implementations in response to changing needs. In this paper, we present one possible formalization of this problem as the task of recovering specifications that locally describe the behavior of individual parts of the circuit, given LTL specifications that globally describe the behavior of the entire circuit. We study the theoretical properties of these *temporal subspecifications*, and show that they are not always expressible in LTL, but can always be described by ω -regular languages. We show that our algorithm can efficiently generate compact subspecifications when applied to benchmarks from the SYNTCOMP 2023 competition. Finally, through a user study, we show that subspecifications improve the accuracy of engineers by a factor of 17 when answering questions about these circuits.

Index Terms—Explainability, linear temporal logic, sequential circuits

I. INTRODUCTION

This paper is about helping engineers understand and debug sequential circuits. Despite a massive amount of research on verification [1], [2], [3], [4], synthesis [5], [6], [7] and repair [8], [9], there has been comparatively less attention given to the task of aiding engineers in debugging, validating and optimizing their designs. There is admittedly some work on helping engineers automatically derive temporal specifications from their code [10], [11], translating these specifications into natural language descriptions [12], and automatically deriving LTL specifications from natural language text [13], [14]. However, these are focused more on the tasks of specification engineering, rather than on helping engineers develop and validate beliefs about different parts of their system.

Indeed, as we will see in our user study, participants struggle to explain the operation of even relatively simple sequential circuits. Although it is easy to obtain execution traces of these systems, design, modification and debugging fundamentally involve reasoning about counterfactual (“*what if?*”) behaviors of different parts of the system.

While studying a similar problem in the context of SyGuS program synthesizers, Nazari et al. [15] proposed the concept of *subspecifications*—i.e., automatically derived specifications of individual subexpressions—as a way of locally explaining what different parts of a loop-free program *should do*. Our

present paper may be alternatively viewed as asking whether a similar notion of subspecifications can be developed in the context of reactive systems.

In our setting, the subspecification corresponds to the set of valid signals that can be produced by individual latches so that the rest of the system satisfies the desired global specification. In other words, subspecifications connect system inputs to possible latch outputs. They therefore provide a way for engineers to characterize the space of valid behaviors of different components, while abstracting away surrounding parts of the system.

The first question that arises when extending the idea of subspecifications to sequential circuits and temporal specifications involves asking what an appropriate language for expressing these temporal subspecs would even be. As we will see in Section IV, it is easy to design circuits where the subspecs for individual components are inexpressible in LTL, even though the global circuit behavior was specified as an LTL formula. We will then show that these subspecs are always ω -regular and may be conveniently expressed as Büchi automata.

We will report on a user study showing that subspecifications massively help users in a range of debugging and validation tasks (improving their response accuracy by 17 $\times$). Finally, we will present an experimental evaluation in which we observe that our algorithm can rapidly derive simple subspecs.

II. FORMALLY DEFINING SUBSPECIFICATIONS

We adapt the following example from the website of the `l1lsynt` tool,¹ distributed as part of the Spot framework [16]. Say an engineer wishes to synthesize a circuit that accepts two Boolean-valued signals i and j as input, and produces an output signal x such that x eventually drops from `true` to `false` iff i and j are both `true` in the initial time step:

$$(i \wedge j) \iff F(x \wedge X\neg x). \quad (1)$$

In response, `l1lsynt` synthesizes the controller shown in Figure 1, both as a state machine and the corresponding and-inverter graph [17], [18].

At this point, say the engineer wishes to understand the purpose of the latch labelled b , with the goal of either optimizing, debugging, or otherwise modifying the circuit. As a first attempt, they might draw a state machine describing the

¹<https://spot.lre.epita.fr/l1lsynt.html>

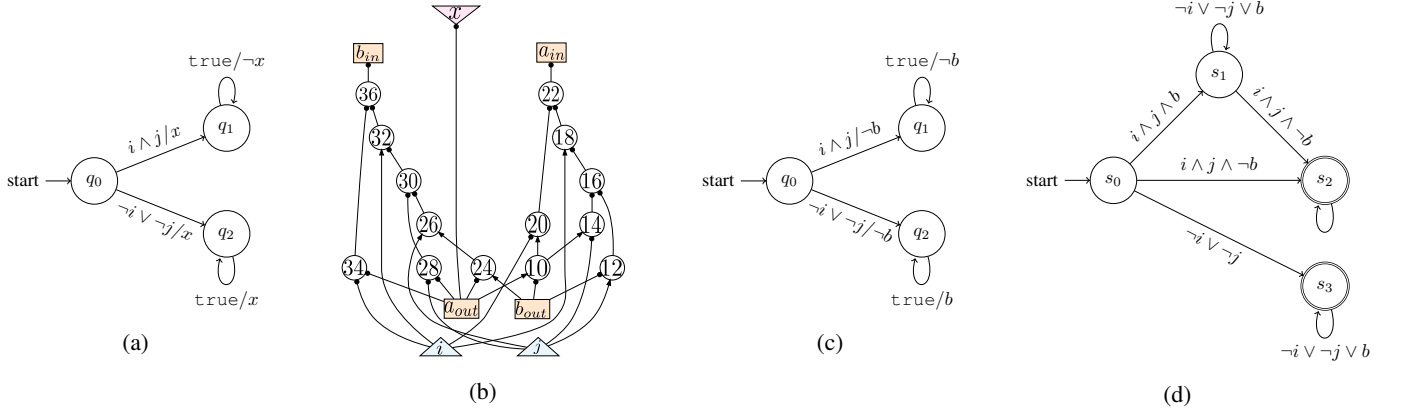


Fig. 1: The controller generated by Spot (ltsynt) for the specification in Equation 1 (1a) and its corresponding and-inverter graph (1b). The numbered circles represent AND gates, and the smaller shaded circles represent inverters. Rectangles represent the input and output sides of latches, and occur in pairs, indicated as X_{in} and X_{out} respectively. Each latch represents a unit time delay and is initialized to *false*. The engineer wishes to know the purpose of the latch b . (1c) Behavior of the latch b from the circuit in Figure 1b. Observe its similarity to the original controller from Figure 1a. (1d) Description of all possible legal behaviors of the latch—i.e., its *subspecification*—so that the entire circuit satisfies Equation 1. Notice that this machine reveals additional possibilities that are not exhibited by the current implementation.

value produced by the latch in response to the history of inputs i and j . See Figure 1c. We observe (unsurprisingly) that this state machine is remarkably similar to the original controller from Figure 1a.

Note however, that although this machine accurately describes the output of latch b , there is a class of questions that it leaves unresolved: For example, if the engineer wishes to optimize or modify this part of the circuit, they would be interested in not just its *current* behavior, but all *possible* behaviors of the latch. They would similarly be interested in possible legal alternative behaviors if changing requirements or faults elsewhere in the circuit need to be mitigated by repairs in this part. Therefore, instead of inquiring about the current behavior of latch b , we are interested in the question: What values *should* the latch b produce, so that the rest of the circuit satisfies the specification in Equation 1?

We can show that b can be replaced by any signal that satisfies the property:

$$(i \wedge j) \iff F(i \wedge j \wedge \neg b). \quad (2)$$

This formula may be equivalently viewed as the Büchi automaton in Figure 1d. Because the future values of i and j are unconstrained, it follows that whenever i and j are *true* in the first time step, b must also produce the initial value *false*. Its output for the rest of time is unconstrained. Alternatively, if either i or j were initially untrue, then it is obligated to obey the constraint $G(i \wedge j \implies b)$. Our central goal with the idea of subspecifications, that we will now formalize, is to provide a uniform answer to counterfactual questions of this kind.

a) Background: Sequential circuits, linear temporal logic (LTL), and Büchi automata: Sequential circuits will form our objects of study in this paper. In brief, a sequential circuit $C = (I, O, L, f)$ is specified by finite sets of Boolean-

valued input and output signals, $I = \{i_1, i_2, \dots, i_m\}$, $O = \{x_1, x_2, \dots, x_n\}$, a finite set of latches, $L = \{a_1, a_2, \dots, a_l\}$, and associated update functions, $f_v : \text{Bool}^{m+l} \rightarrow \text{Bool}$, for each $v \in O \cup L$.

The inputs supplied to the circuit may be modeled as an infinite sequence of valuations, $\sigma = \sigma_1, \sigma_2, \dots$, of each of the input signals I . The circuit responds by iteratively computing the values of its latches, ρ , and output signals, τ , as follows:

$$\begin{aligned} \rho_0(a) &= \text{false}, \\ \rho_{i+1}(a) &= f_a(\sigma_i, \rho_i), \text{ and} \\ \tau_i(x) &= f_x(\sigma_i, \rho_i), \end{aligned}$$

for $a \in L$, $x \in O$, and $i \in \mathbb{N}$. For example, the circuit in Figure 1b may be represented using the set of update functions:

$$\left. \begin{aligned} a_{n+1} &= \bar{i}_n a_n \bar{b}_n + i_n \bar{j}_n a_n \bar{b}_n + i_n j_n \bar{b}_n, \\ b_{n+1} &= \bar{i}_n \bar{a}_n + i_n \bar{j}_n \bar{a}_n + i_n j_n \bar{a}_n b_n, \text{ and} \\ x_n &= \bar{a}_n. \end{aligned} \right\} \quad (3)$$

Because of limited column width, we will sometimes use $(+, \cdot, \bullet)$ notation instead of $(\vee, \wedge, \neg)$.

We specify properties of these sequential circuits using formulas in LTL. Recall that an LTL formula ϕ is a production of the grammar:

$$\begin{aligned} \phi &::= \text{true} \mid \text{false} \mid v \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \\ &\quad \mid X\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2, \end{aligned}$$

where $v \in I \cup O \cup L$. The interpretation of these formulas over infinite traces is standard. We refer the reader to Clarke et al.'s textbook on model checking [19].

One may alternatively specify properties of infinite signals using Büchi automata. A Büchi automaton is a structure $M = (Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite

alphabet (in our case, most commonly Bool^{m+n}), $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. The machine in Figure 1d is an example. We say that the machine accepts an ω -string $w = w_1, w_2, \dots$ if there exists a corresponding run $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \xrightarrow{w_3} \dots$ in which some state $q_f \in F$ occurs infinitely often. We refer to the language that M accepts as $L(M) \subseteq \Sigma^\omega$. Once again, we refer the reader to [19].

b) Sequential subspecifications: Let $C = (I, O, L, f)$ be a sequential circuit, and let ϕ be an LTL specification with variables from $I \cup O$. Let $b \in L$ be a latch that the engineer wishes to investigate. We can use C to construct a new circuit:

$$C|_b = (I \cup \{b\}, O, L \setminus \{b\}, f \setminus \{b \mapsto f_b\}).$$

Informally, this amounts to promoting b , currently calculated by a latch, to the status of a new input, while leaving the rest of the circuit unchanged. We say that an LTL formula ψ is a *subspecification* of the latch b with respect to the global specification ϕ if for each sequence of inputs $\tilde{\sigma}$ supplied to $C|_b$, we have:

$$\tilde{\sigma} \models \psi \iff (\sigma, \tau) \models \phi, \quad (4)$$

where τ is the corresponding sequence of outputs produced by $C|_b$, and σ is the projection of $\tilde{\sigma}$ obtained by eliminating the values of b .

Note II.1. 1) *Informally:* The subspec ψ specifies all possible alternative values that could have been produced by b , so that the rest of the circuit $C|_b$ still satisfies ϕ .

- 2) Also note that we do not require the initial circuit C to itself satisfy ϕ . This flexibility is useful when engineers are concerned with problems of debugging and repair, as we will see in the example in Section III-A.
- 3) By considering sequential circuits and temporal properties rather than stateless expressions, the definition of subspecs in Equation 4 is a strict generalization of the idea initially developed by [15].
- 4) As discussed by [15], it is possible to introduce the idea of joint subspecs which characterize how subspecs of multiple components interact and compose. However, it requires technical assumptions, see Theorem 5.7 of [15].

Ideally, one would like to express the specification and subspec in the same language. However, it is easy to construct circuits where the subspec is inexpressible as an LTL formula. We will see an example in Section IV-A. We will therefore primarily be interested in situations where the subspec is represented as a Büchi automaton. We say that a Büchi automaton M with alphabet $\Sigma = \text{Bool}^{I \cup \{b\}}$ is the subspecification of b with respect to ϕ if:

$$\tilde{\sigma} \in L(M) \iff (\sigma, \tau) \models \phi,$$

where (as before) τ is the sequence of outputs produced by $C|_b$, and σ is the sequence of valuations obtained from $\tilde{\sigma}$ by projecting out the values of the “real” inputs, $i \in I$.

c) Paper outline: In Section III, we will present two additional examples illustrating the utility of subspecs for validation and debugging. Then, in Section IV, we will present an algorithm to automatically derive these subspecifications. Finally, in Sections V and VI, we will focus on empirically validating their usefulness and our effectiveness in producing compact subspecs.

III. EXAMPLE APPLICATIONS

Our examples in this section will be drawn from the Reactive Synthesis Competition, SYNTCOMP 2023 [6]. Like for the example in Section II, automatically synthesized controllers provide a convenient source of specifications and implementations that are tricky to comprehend.

A. Debugging Circuits

We consider the example of `lilydemo12.tlsf` [20]. The original goal was to synthesize a controller that maps a pair of input signals, i, j , to a pair of output signals, x and y , such that:

$$G \neg x \vee G(i \implies Fy) \vee G(j \implies Fx).$$

In response, Spot produces a controller implemented using the following two-latch circuit:

$$\left. \begin{aligned} a_{n+1} &= \bar{j}_n \bar{a}_n \bar{b}_n, \\ b_{n+1} &= j_n \bar{a}_n + \bar{a}_n b_n + a_n \bar{b}_n, \\ x_n &= \bar{j}_n a_n \bar{b}_n + \bar{a}_n b_n, \text{ and} \\ y_n &= i_n \bar{a}_n + i_n \bar{b}_n. \end{aligned} \right\} \quad (5)$$

Assume that an engineer mistakenly designs the circuit as follows:

$$x'_n = \underbrace{\neg x_n} \quad \text{and} \quad y'_n = (\underbrace{\neg i_n} \wedge \neg a_n) \vee (\underbrace{\neg i_n} \wedge \neg b_n). \quad (6)$$

In other words, two mistakes were made: the output x was incorrectly negated, and the input i was incorrectly negated while being used to compute y . Note that these mistakes correspond to two bit flips in the AIGER-encoded circuit.

At this point, the engineer might wish to explore ways of repairing the system. Among other questions, they might wonder whether its functionality can be restored by changing the values produced by latch b . Although one might draw the state machine corresponding to the current computation of b in a manner similar to what we did in Figure 1c—see Figure 2a—we note that this is useless, because we are uninterested in what b *currently does*, and instead interested in what the latch *should now be doing*.

As part of our user study in Section V, we asked a group of students to suggest possible ways of repairing the circuit by modifying the behavior of b . Notably, without additional assistance, only one participant of nine was able to solve the task, and required approximately 8 minutes to identify a fix.

Alternatively, using our subspecification derivation algorithm from Section IV-B, one discovers that the latch can be replaced with any component all of whose behaviors are accepted by the Büchi automaton shown in Figure 2b. Observe now that

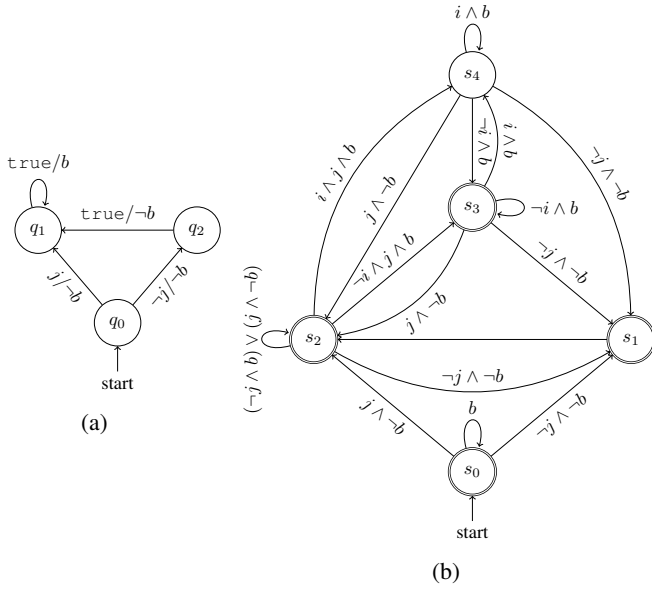


Fig. 2: (2a) The controller for the latch b in the circuit of Equation 5. (2b) A description of all possible behaviors of the latch b so that the rest of the faulty circuit from Equation 6 nevertheless satisfies the specification in Equation III-A.

the erroneous circuit would satisfy the specification if b were to simply be replaced with a signal that produces the constant value `true`. By replacing value of b in Equation 5 with this new signal, $b'_n = \text{true}$, one observes that it results in the new sequence of output values $x''_n = \text{false}$ and $y''_n = \neg i_n$. This repaired implementation would therefore satisfy the specification by fulfilling its leftmost term, $G\neg x$.

Eight of the 9 participants in the intervention group in our user study suggested this method of fixing the system. The remaining participant identified the following (only slightly more complicated) fix. They observed that state q_3 was the only non-accepting state in the subspec automaton in Figure 2b and pointed out that all transitions leading to this state would be disabled if $b'_n = \neg i_n$. Plugging in this fix into the faulty update expressions in Equation 6 and simplifying reveals that, in this case, $y''_n = i_n$, so that the repaired implementation works by fulfilling the second term in the specification, $G(i \implies Fy)$.

B. Validating Sequences

We now look at `example72.tlsf` from the SYNTCOMP 2023 benchmark suite. Here, we are interested in a two-input (i, j) two-output (x, y) controller such that:

$$G(\neg x \vee \neg y) \wedge G(i \implies x \vee Xx) \wedge G(j \implies y \vee Xy). \quad (7)$$

The two-latch circuit in question is specified by the following update expressions:

$$\left. \begin{aligned} a_{n+1} &= i_n j_n \bar{a}_n + \bar{i}_n j_n \bar{a}_n \wedge b_n, \\ b_{n+1} &= a_n \bar{b}_n \wedge i_n, \\ x_n &= \bar{i}_n j_n \bar{a}_n b_n + \bar{i}_n j_n \bar{a}_n + i_n \bar{a}_n, \text{ and} \\ y_n &= i_n \bar{b}_n a_n + \bar{i}_n j_n \bar{b}_n + \bar{i}_n j_n a_n \bar{b}_n. \end{aligned} \right\} \quad (8)$$

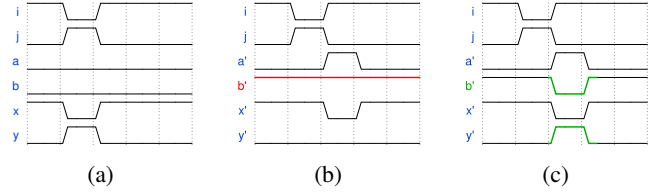


Fig. 3: (3a) Original behavior of the system resulting from signal i turning off and j turning on for one clock cycle in the second time step. The engineer wonders why the circuit would not work if b were to produce the constant value `true`. (3b) The values produced by a' , x' and y' in this counterfactual scenario. This execution trace fails the specification because y' never goes high in response to the impulse on j . (3c) Analyzing the subspec for b reveals that pushing it to false in the third time step would restore global correctness.

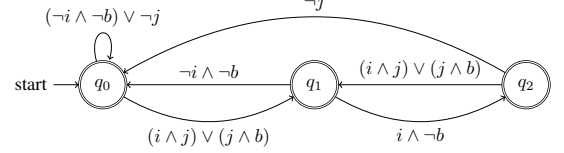


Fig. 4: A description of all possible legal behaviors of the latch b so that the circuit from Equation 8 satisfies Equation 7.

As part of their validation process before incorporating this system in their designs, the engineer might simulate the controller under a variety of test inputs. We show an example trace in Figure 3a. In this situation, they observe that both latches a and b uniformly remain at `false`. They ask whether the circuit still works if b is forced to be constantly `true`. Questions like this might conceivably also arise when they are modifying the circuit and running test cases.

Modifying the behavior of latch b in this manner would affect the computation both of the remaining latch a and of the outputs x , y , resulting in the alternative execution trace shown in Figure 3b. It can be seen that this trace does not satisfy the specification in Equation 7.

We now observe that subspecifications can provide greater insight into the causes for this failure. We show the automatically calculated subspec automaton in Figure 4. It turns out that the new trace for b causes this machine to pass through the sequence of states $q_0 \rightarrow q_0 \rightarrow q_1$. At this point, the run would terminate because both outgoing transitions from q_1 are disabled. This analysis allows us to localize the fault within this alternative trace to the third time step. Pushing b to false for one clock cycle at this point would cause the resulting trace to once again satisfy the global specification. See Figure 3c.

IV. EXPRESSIBILITY AND AUTOMATIC DERIVATION OF SUBSPECS

The principal contribution of this paper over [15] is in extending the idea of subspecifications from the setting of stateless, loop-free expressions to the more general setting of sequential circuits. Naturally, we need to reconsider questions related to

expressiveness and develop new algorithms to automatically derive these subspecs (if they exist). Unfortunately, it is easy to show that even if the global specification is provided as an LTL formula, the subspecification of a latch need not itself always be expressible using LTL. See Theorem IV.1. On the other hand, in Section IV-B we show that the subspec is always expressible as a Büchi automaton. Our proof of this second result also provides an algorithm to automatically derive these subspecifications.

A. Inexpressibility of Subspecifications as LTL Formulas

We start with the simple two-latch circuit shown in Figure 9a in Appendix A. Each latch flips its value from the previous time step:

$$a_{n+1} = \neg a_n \quad \text{and} \quad b_{n+1} = \neg b_n. \quad (9)$$

Recall that both latches are initialized as $a_0 = b_0 = \text{false}$. The circuit calculates its single output bit as follows:

$$x_n = a_n \vee \neg b_n.$$

Naturally, this circuit always produces the output `true`, thereby satisfying Gx . We will now show that:

Theorem IV.1. *There is no LTL formula which describes the subspecification of latch b in the circuit of Figure 9a in Appendix A and with respect to the global specification, Gx .*

Proof. It is easy to see that for this global property, Gx to hold, the latch b must produce the value `false` in even time steps, $t = 0, 2, 4, \dots$. Its value in odd time steps, $t = 1, 3, 5, \dots$, is unconstrained. This subspecification may be represented using the Büchi automaton in Figure 9b, but is famously inexpressible as an LTL formula [21]. $\square$

B. Automatically Deriving Subspecs as Büchi Automata

We will now describe an algorithm to obtain the subspecification of a latch b when it is requested as a Büchi automaton. Recall that the problem is to replace the latch with a new “magic” input signal, and determine all possible sequences of inputs that can be supplied to this new circuit $C|_b$ so that the execution of the rest of the circuit satisfies the given global specification ϕ . For the purpose of illustration, we will continue with the example from Section II.

As a first step, we write down an LTL formula that describes all possible executions of the circuit $C = (I, O, L, f)$:

$$\chi(C) = \bigwedge_{x \in O} G(x \iff f_x(I, L)) \wedge \bigwedge_{a \in L} (\neg a \wedge G(Xa \iff f_a(I, L))). \quad (10)$$

This formula, $\chi(C)$, ranges over the free variables $I \cup O \cup L$, and functions in a manner similar to the Tseitin transform [22]:

Lemma IV.2. *A sequence of valuations (σ, τ, ρ) of $I \cup O \cup L$ satisfies $\chi(C)$ iff the circuit C produces the output sequence τ and latch values ρ when provided with the input sequence σ .*

Proof. By induction on the timestep i and repeated application of the circuit semantics from Section II. $\square$

Observe that $\chi(C|_b)$ therefore describes all possible executions of the promoted circuit $C|_b$. For example, for the circuit in Equation 3, $\chi(C|_b)$ would be:

$$G(x \iff \bar{a}) \wedge \bar{a} \wedge G(Xa \iff \bar{i}a\bar{b} + \bar{i}j\bar{a}\bar{b} + \bar{i}j\bar{b}).$$

We are interested in executions of $C|_b$ that also satisfy ϕ . Naturally, the formula of interest is $\chi(C|_b) \wedge \phi$. This formula can be readily transformed into an equivalent Büchi automaton $M_{C,b,\phi}$. Figure 8 in Appendix A shows the resulting construction when applied to our running example. The main outstanding challenge is that $\chi(C|_b) \wedge \phi$ (and therefore $M_{C,b,\phi}$) ranges over all variables, $I \cup O \cup L$, while the desired subspec in Figure 1d only relates the values of the inputs I and the latch b that is currently being investigated.

Our key insight is that because every execution of $M_{C,b,\phi}$ corresponds to an execution of $C|_b$, we can obtain the subspec by simply erasing the irrelevant fields of the input alphabet, $\Sigma = \text{Bool}^{I \cup O \cup L}$. Let $M_{C,b,\phi} = (Q, \Sigma, \Delta, q_0, F)$. Formally, we propose to construct the projected-down automaton,

$$M_{C,b,\phi}^\downarrow = (Q, \Sigma^\downarrow, \Delta^\downarrow, q_0, F), \quad (11)$$

by selecting the fields in Σ corresponding to $I \cup \{b\}$, so that $\Sigma^\downarrow = \text{Bool}^{I \cup \{b\}}$ and

$$\Delta^\downarrow = \{(q, a^\downarrow, q') \mid (q, a, q') \in \Delta\},$$

and where $a^\downarrow \in \text{Bool}^{I \cup \{b\}}$ is the symbol obtained by eliminating the unnecessary field of $a \in \text{Bool}^{I \cup O \cup L}$. We can now establish a correspondence between the executions of $M_{C,b,\phi}$ and the executions of $M_{C,b,\phi}^\downarrow$:

Lemma IV.3. *Whenever the ω -path $\pi = q_0 \rightarrow^{a_0} q_1 \rightarrow^{a_1} q_2 \rightarrow \dots$ is accepted by $M_{C,b,\phi}$, the corresponding path $\pi^\downarrow = q_0 \rightarrow^{a_0^\downarrow} q_1 \rightarrow^{a_1^\downarrow} q_2 \rightarrow \dots$ is also accepted by $M_{C,b,\phi}^\downarrow$. Conversely, for every path $\pi^\downarrow$ accepted by $M_{C,b,\phi}^\downarrow$, there exists a path π accepted by $M_{C,b,\phi}$ such that $\pi^\downarrow$ is the projection of π .*

Proof. The forward direction is immediate. In the converse direction, recall that every transition $(q, a^\downarrow, q') \in \Delta^\downarrow$ corresponds to some transition $(q, a, q') \in \Delta$ of $M_{C,b,\phi}$. For each transition $q_i \rightarrow^{a_i^\downarrow} q_{i+1}$ in $\pi^\downarrow$, arbitrarily pick $a_i \in \Sigma$ so that $(q_i, a_i, q_{i+1}) \in \Delta$. It must be the case that $\pi = q_0 \rightarrow^{a_0} q_1 \rightarrow^{a_1} q_2 \rightarrow \dots$ is a valid path through $M_{C,b,\phi}$. Furthermore, because $M_{C,b,\phi}^\downarrow$ preserves the acceptance conditions, it must be the case that π is also accepted by $M_{C,b,\phi}$, thus completing the proof. $\square$

Note IV.4. Recall that because $C|_b$ is deterministic, it is possible to uniquely recover the values of all latches $a \in L$ and output signals in $x \in O$ by simply examining the history of the values $i \in I \cup \{b\}$. Furthermore, because every path $\pi^\downarrow$ through $M_{C,b,\phi}^\downarrow$ corresponds to some path π of $M_{C,b,\phi}$, and every execution of $M_{C,b,\phi}$ corresponds to an execution of $C|_b$, the reconstruction

guaranteed by Lemma IV.3 must be unique: i.e., there must be a unique reconstructed path π for each $\pi^\downarrow$.

Combining Lemmas IV.2 and IV.3, we have:

Theorem IV.5. *For each circuit $C = (I, O, L, f)$, specification ϕ , and latch $b \in L$, the Büchi automaton $M_{C,b,\phi}^\downarrow$ is a subspecification of b with respect to ϕ .*

Proof. Recall that $C|_b$ is the circuit obtained by promoting the latch b to the status of a new input signal, and that $M_{C,b,\phi}$ is the Büchi automaton accepting the same set of traces as $\chi(C|_b) \wedge \phi$. Therefore, by Lemma IV.2, $M_{C,b,\phi}$ accepts exactly those traces of $C|_b$ that also satisfy ϕ . From Lemma IV.3, there is a correspondence between the traces of $M_{C,b,\phi}$ and $M_{C,b,\phi}^\downarrow$, so it follows that $M_{C,b,\phi}^\downarrow$ accepts only those signal traces such that the reconstructed execution of $C|_b$ would satisfy the specification ϕ . Because $M_{C,b,\phi}^\downarrow$ only considers the values of the extended set of input signals, $I \cup \{b\}$, we can now say that $M_{C,b,\phi}^\downarrow$ is the subspec of latch b with respect to the specification ϕ . The theorem follows. $\square$

a) Implementation details: (a) We use Owl [23] for the LTL-to-Büchi automaton translation, and the `autfilt` tool in Spot [16] for simplifying the resulting automata. (b) We use Spot to translate transition guards into DNF form. The minterms of this formula can be easily subject to the downward projection operation. (c) Although the definition of subspecs in Section II focused on latches, our implementation more generally allows for the computation of subspecs for any component in the AIGER-encoded circuit. Notice that defining subspecs for these components is an easy generalization. In particular, if these Boolean functions are themselves representable using combinational circuits, then we can perform similar reasoning with any of its internal components as well. In particular, we can replace the output of each internal logic gate b with a hypothetical new input signal, and consider the conditions under which the revised circuit would satisfy the global specification. This would constitute a minor extension of the more restricted idea of subspecs as presented in this paper, but the computation algorithm would remain the same: Construct $C|_b$, and use this to calculate (in sequence,) $\chi(C|_b)$, $M_{C,b,\phi}$, and $M_{C,b,\phi}^\downarrow$, and finally simplify.

V. EMPIRICALLY MEASURING THE UTILITY OF SUBSPECIFICATIONS

The first part of our evaluation consisted of a user study to determine whether subspecifications were helpful to engineers. Our goal was to answer the following research questions:

- RQ1.** Do subspecs help users in distinguishing valid and invalid execution traces?
- RQ2.** Do subspecs help users in explaining the purpose of individual components?
- RQ3.** Do subspecs help users in repairing faulty circuits?

A. Participants, Tasks, and Study Structure

a) Participant selection and screening process: The study was conducted after obtaining IRB approval. We recruited

18 graduate students (2 Masters and 16 Ph.D. students) from the Computer Science (CS), Electrical Engineering (EE), and Industrial and Systems Engineering (ISE) departments of two prominent American and Canadian universities. These participants had a range of specializations, including optimization algorithms, human-computer interaction, machine learning and natural language processing, software engineering, MEMS and robotics, computer networking, and theoretical CS.

We started by providing the participants with an introduction to the study, and briefly introducing them to temporal logic and the idea of subspecifications. We then administered a screening quiz with 5 questions to ensure that participants had a baseline level of understanding of these background ideas. All participants received perfect scores in the screening quiz, and were therefore included in the main study.

b) Tasks and study structure: The study consisted of three tasks. The first task was based on the specification-implementation pair discussed in Section III-B. It consisted of 4 questions in which participants were asked to predict whether a presented counterfactual trace would cause the rest of the circuit to produce an output trace that satisfied the specification. We also asked participants to justify their responses. The second task built on our introductory example in Section II, and asked participants to explain, in natural language, the constraints that specific parts of the circuit must satisfy. The last task involved the faulty system discussed in Section III-A. We pointed participants to different parts of the circuit, and asked them to suggest fixes. We also asked them to justify their responses if implementing a repair was impossible.

The study was formulated as a repeated measures design, i.e., one in which each participant attempted at least one task with access to subspecs and at least one other task without access to subspecs. For each task, participants were randomly assigned to either the intervention or control arms, with exactly 9 participants attempting each task under each condition. The screening quiz, study materials, and (anonymized) participant responses will be included as part of our artifact.

All authors of this paper independently graded participant responses. The pairwise correlation coefficients between our grades were 0.85, 0.88 and 0.90 respectively. We present our average grades (indicating our assessment of their accuracy) in Figure 6.

B. RQ1: Distinguishing Valid and Invalid Execution Traces

We draw our conclusions from Questions 1.1–1.4 of the user study. We expected each response to include both a summary Boolean-valued judgment (“*Counterfactual trace leads to valid behavior*” vs. “*Counterfactual trace leads to erroneous behavior*”) and a justification.

Two trends are obvious from Figure 6a: Participants without access to subspecs were broadly unsuccessful at the task while participants with access to subspecs were significantly more effective. With and without our intervention, average participant accuracy was 94% and 3% respectively.

In the baseline-without-subspecc condition, participants would have had to first mentally simulate the circuit from the provided

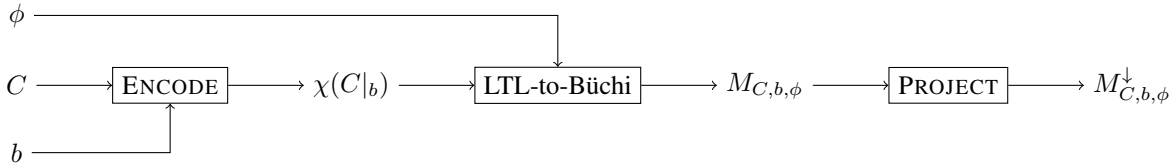


Fig. 5: Schematic overview of the algorithm to derive subspecifications. The procedure takes as input the global specification ϕ , the circuit C , and the latch b whose subspec is being queried. We use Equation 10 to convert $C|_b$ into a corresponding LTL formula and obtain $M_{C,b,\phi}$ by using a standard LTL-to-Büchi automaton translator. Next, we obtain the subspec $M_{C,b,\phi}^\downarrow$ using the construction described in Equation 11 and subsequently simplify.

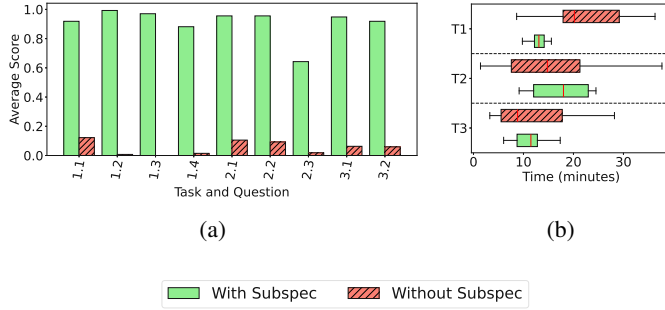


Fig. 6: Accuracy of participant responses and distribution of time needed to complete each of the study tasks. Questions 1.1–1.4 assessed the ability of participants to validate counterfactual traces, Questions 2.1–2.3 asked them to explain what different components should do, and Questions 3.1 and 3.2 assessed their ability to repair faulty circuits.

update equations, and then determine whether the induced response satisfied the provided specification. Anecdotally, given the complexity of the update equations, most participants were unable to even simulate the circuit. In the post-study debrief, several of these participants complained about the complexity of the update equations.

Accordingly, we observed two distinct response patterns from the control group: The first subgroup opted to skip the question after spending a considerable amount of time (Notice the massively larger length of time that these participants spent on Task 1), while the second subgroup provided guesses without sound reasoning or justification. We also noticed that participants gradually became tired, so their response accuracy for Q1.1 was noticeably higher than for the remaining questions.

In contrast, the subspec eliminated the need to mentally simulate the circuit. Participants simply had to trace the behavior of the subspec automaton in response to the inputs and counterfactual latch values. The subspec therefore allowed the participants to visualize and locally reason about the execution trace, without having to engage with the rest of the circuit’s components.

C. RQ2: Explaining the Purpose of Individual Components

Now, we draw our conclusions from observing participant responses to Questions 2.1–2.3. Specifically, these questions

asked participants to describe the required behavior of individual latches so that the rest of the circuit satisfied the specification. As a point of elaboration, we asked participants for the considerations that designers must keep in mind while modifying the implementation.

Notice that, unlike the first task (which admitted a clear solution strategy even without access to subspecs,) this second task was open-ended. Here, most participants in the control group confessed to not even knowing where to start. Participants who had access to subspecs tended to approach the problem by performing a case analysis on the subspec automaton.

The components of interest in Questions 2.1 and 2.2 admitted relatively simple subspecs which only constrained their behavior in the initial time step. Consequently, all participants had a relatively higher accuracy for these two questions. In contrast, Question 2.3 was exactly the setting of latch b that we examined while initially motivating subspecs in Section II.

In this case, subspecs made some behaviors obvious: In particular, if $i \wedge j$ was *false* in the initial time step, then b was required to satisfy $\mathbf{G}(i \wedge j \implies b)$. Similarly, if $i \wedge j \wedge \neg b$ held in the initial time step, then all requirements were lifted for the rest of time. On the other hand, if $i \wedge j$ was true in the first time step, and b also assumed the value *true*, then the subspec automaton would transition to the state s_1 , which did not admit a winning strategy. It was therefore crucial for the latch b to produce the initial value *false* when initially $i \wedge j$. Four of the nine participants who had access to subspecs (and nobody in the control group) were able to completely articulate this requirement.

D. RQ3: Repairing Faulty Circuits

Finally, we focus on our observations of participant responses to Questions 3.1 and 3.2. Both these questions involve the specification-implementation pair from Section III-A.

Once again, participants in the control group complained about having insufficient information to complete the task. We also noticed them becoming tired: after spending considerable effort and still being unsuccessful in Question 3.1, some of them chose to skip Question 3.2.

While designing the user study, we expected this to be the hardest of the three tasks. We were surprised that participants with access to subspecs achieved an average score of 93%, and needed the least amount of time among all three tasks. Another notable observation was that the circuit could not be repaired

by modifying the component highlighted in Question 3.2. We show its subspec in Figure 10 in Appendix A. For this question, the average score of participants in the intervention group was 92%, indicating that most of them successfully identified and justified the unrepairability of component m .

VI. EFFECTIVENESS OF THE SUBSPEC GENERATION PROCEDURE

Next, we measured the effectiveness of our algorithm for deriving simple subspecifications. We were interested in two research questions:

RQ4. Does the algorithm generate “*simple*” subspecs?

RQ5. How long does the procedure take to construct these subspecs?

A. Benchmarks

We ran Strix [7], the winner of the SYNTCOMP 2023 Competition on all benchmark specifications used in the competition. We collected the generated controllers and corresponding circuit implementations. We set a 5 minute timeout on the synthesizer, within which the solver was able to synthesize 635 controllers. Recall from the discussion in Section IV-B that our tool is able to calculate subspecs for not just latches, but more generally, for any component in an AIGER-encoded circuit. We focused on controllers which had less than 100 such components, resulting in 545 specification-implementation pairs, and which collectively contained 13,208 components. We ran the subspec generation tool on each of these components with a timeout of 10 minutes per run. At the end of this data collection process, we had access to subspecs for 11,453 components.

B. RQ4: Effectiveness in Simplification

We measured the sizes of the generated subspecs. As such, one would expect that the size of these subspecs is dependent on the complexity of the specification or the controller being investigated. Therefore, in Figures 7a and 7b, we present the distributions of subspec size (# of states) when compared to the size of the original specification (# of AST nodes) and the size of the implementation respectively.

We notice that as many as 46% of the components in question admit subspecs that are less than 20% of the size of the original specification. Furthermore, in 75% of the cases, the subspec is smaller than the original specification. In only 7% of the cases is the subspec $> 10\times$ of the size of the original specification. We make similar observations when comparing the size of the subspec to the size of the circuit that surrounds the component of interest: in this case, the corresponding numbers are 22%, 55%, and 6% respectively. Of course, all these comparisons need to be interpreted with some care, because of the different units of measurement associated with the subspec and the original specification / implementation.

Nevertheless, we may broadly conclude that our algorithm is effective in generating simple subspecifications. We also note that we post-process the subspec initially produced by our procedure using the automata simplification routine implemented

in Spot’s `autfilt` tool. Figure 7c shows measurements of the effectiveness of this simplification procedure. It achieves a $\geq 50\%$ compression in 50% of all cases. This appears to be because most of the states in the originally constructed automaton, $M_{C,b,\phi}$ reason about other parts of the circuit, and are useless after the downward projection into $M_{C,b,\phi}^\downarrow$. We therefore believe that such post-processing passes are important in obtaining simple subspecs.

C. RQ5: Time Needed to Derive Subspecifications

We present the running time measurements of the subspec generation procedure in Figures 7d and 7e: These figures respectively describe the absolute running time and a comparison to the time needed to synthesize the original controller.

Note that 57% of cases require less than a second for subspec generation, and we are faster than the original synthesis run in 66% of cases. Only 5% of cases require long periods of waiting. Our long-term goal is for engineers to consult subspecs interactively during system design. The algorithm’s current performance appears adequate for this.

VII. RELATED WORK

a) Verification and synthesis of reactive systems: Automatic verification and synthesis are foundational and widely studied problems [24]. Numerous algorithms and tools have been proposed [7], [4], with annual competitions highlighting recent advances [3], [6]. Verification tools aid reliable system design by finding bugs and certifying correctness. Indeed, one of the important attractions of model checking is its ability to generate counter-example traces when the system fails to satisfy the desired property [25]. However, these counter-example traces describe executions of the entire system, and are not immediately helpful in localizing the fault or in devising repairs. As such, these are not questions about the *current* behavior of the system, but rather, of its *desired* behavior.

b) Deriving and explaining LTL specifications: There has also been some concern about the inaccessibility of formal specification languages for engineers in applied fields like robotics, who may lack expertise in verification. To address this, some efforts have aimed to make these formalisms more accessible: for example, by automatically deriving temporal logic specifications from system models [10], [11], translating LTL formulas into natural language descriptions [12], and using various kinds of translation technology to convert requirements expressed in natural language into LTL, STL, MTL, and other kinds of temporal logic formulas [13], [14]. Of course, these techniques focus more on issues of understanding and obtaining good specifications, rather than on the task of explaining the mechanics of the system under consideration.

c) Modular verification and local reasoning: The key idea in this paper was to reverse-engineer (temporal) specifications for individual components in a composite system. As such, this task is intimately tied to the problem of modular verification [26]. The promise of modular verification is that proving and composing component-level properties can lead to more scalable verification. In a sense, our hope with subspecs

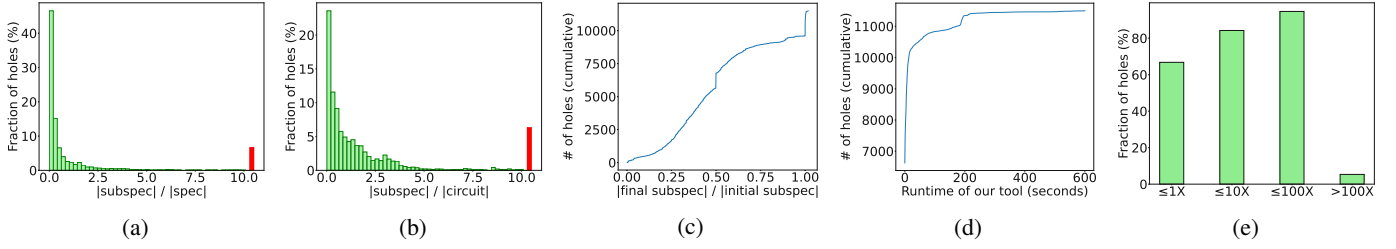


Fig. 7: (7a) Distribution of the size of the generated subspec (measured as the number of states in the subspec automaton), in comparison to the size of the original specification (measured as the number of AST nodes). By *holes* we are referring to the fraction of target components with a given $|subspec|/|spec|$ ratio. The red colored bar indicates cases where the subspec size was $> 10\times$ of the specification. (7b) Distribution of subspec size when compared to the size of the controller (measured as the number of components in the AIGER implementation). (7c) Effectiveness of the subspec simplification pass described in Section IV-B. Cactus plot of the subspec derivation time (7d) and comparison to the time needed for originally synthesizing the circuit (7e).

is the same: that engineers will find properties of specific parts intuitive and easy-to-reason about when isolated from the rest of the system. Of course, one challenge with modular verifiers is inferring properties of individual modules. Similar challenges might also arise when applying subspecs to very large systems.

d) The unknown component problem: Subspecs are formally similar to the quotient operation over assume-guarantee contracts [27], [28]: Given a global specification $C = (A, G)$ and a contract for the rest of the implementation $C' = (A', G')$, the quotient, C/C' is the weakest specification of the unknown component [29] C'' such that their composition, $C' \otimes C''$, satisfies C . Incer et al. [27] explicitly characterize the quotient as $C/C' = (A \cap G', A' \cap G \cup \neg(A \cap G'))$. The principal difference is that this formula ranges over all variables in the system, while the subspec (obtained by downward projection in Equation 11) constrains the behavior of the unknown component with respect to only the global system inputs. In a sense, quotients correspond to the control group in our user study, where participants had access to the actual implementation of the rest of the system. We have therefore demonstrated the value of information hiding (i.e., of erasing irrelevant variables from the subspec) during system engineering tasks.

e) Program comprehension and repair: Subspecs are also closely connected to the problem of program repair. Most simply, the behaviors exhibited by the program patch must

satisfy the corresponding subspec. Program repair has been extensively studied, both in the setting of large-scale code [30], [31] and in the setting of reactive systems [32]. Another notable body of research focuses on program comprehension: one approach involves sophisticated techniques to visualize program executions [33], while the other—for e.g., the famous Whyline tool [34]—once again involves counterfactual questions.

VIII. CONCLUSION

We introduced temporal subspecifications, a new way to explain the behavior of individual components in sequential circuits. Our algorithm efficiently extracts these localized specifications, often producing simpler representations than the original design. In a user study, subspecs greatly improved engineers' accuracy in understanding, debugging, and repairing circuits. This approach opens the door to more intuitive, component-level reasoning and future integration into practical circuit design and debugging tools.

ARTIFACT AVAILABILITY STATEMENT

The artifact supporting the claims made in this paper may be downloaded from Zenodo [35].

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under Grants CCF-2146518 and CCF-2107261.

REFERENCES

- [1] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*. Springer, 1981, pp. 52–71.
- [2] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *International Symposium on programming*. Springer, 1982, pp. 337–351.
- [3] A. Biere, N. Froleyks, and M. Preiner, "Hardware model checking competition 2024," in *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD, 2024.
- [4] A. Goel and K. Sakallah, "Avr: abstractly verifying reachability," 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. Springer, 2020, pp. 413–422.
- [5] 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*, 1989, pp. 179–190.
- [6] S. Jacobs, G. Perez, and P. Schlehuber-Caissier, "Data, scripts, and results from SYNTCOMP 2023," 2023. [Online]. Available: <https://doi.org/10.5281/zenodo.8161423>
- [7] P. J. Meyer, S. Sickert, and M. Luttenberger, "Strix: Explicit reactive synthesis strikes back!" in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 578–586.
- [8] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings* 17. Springer, 2005, pp. 226–238.
- [9] A. Griesmayer, R. Bloem, and B. Cook, "Repair of boolean programs with an application to c," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings* 18. Springer, 2006, pp. 358–371.
- [10] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 81–92.
- [11] D. Neider and R. Roy, "What is formal verification without specifications? a survey on mining ltl specifications," in *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*. Springer, 2024, pp. 109–125.
- [12] H. Cherukuri, A. Ferrari, and P. Spoletini, "Towards explainable formal methods: From ltl to natural language with neural machine translation," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2022, pp. 79–86.
- [13] F. Fuggitti and T. Chakraborti, "NL2LTL: A Python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas," in *AAAI*, 2023, system Demonstration.
- [14] —, "NL2LTL: A Python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas," in *ICAPS*, 2023.
- [15] A. Nazari, Y. Huang, R. Samanta, A. Radhakrishna, and M. Raghothaman, "Explainable program synthesis by localizing specifications," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, 2023. [Online]. Available: <https://doi.org/10.1145/3622874>
- [16] 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)*, ser. Lecture Notes in Computer Science, vol. 13372. Springer, 2022, pp. 174–187.
- [17] A. Biere, "The AIGER And-Inverter Graph (AIG) format version 20071012," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 07/1, 2007.
- [18] S. Jacobs, "Extended AIGER format for synthesis," *CoRR*, vol. abs/1405.5793, 2014. [Online]. Available: <http://arxiv.org/abs/1405.5793>
- [19] E. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*, 2nd ed. MIT Press, 2018.
- [20] B. Jobstmann and R. Bloem, "Optimizations for ltl synthesis," in *Formal Methods in Computer Aided Design*, ser. FMCAD, 2006, pp. 117–124.
- [21] P. Wolper, "Temporal logic can be more expressive," *Information and Control*, vol. 56, no. 1, pp. 72–99, 1983. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0019995883800515>
- [22] G. S. Tseitin, "On the complexity of derivation in propositional calculus," 1983. [Online]. Available: <https://api.semanticscholar.org/CorpusID:123007433>
- [23] J. Křetínský, T. Meggendorfer, and S. Sickert, "Owl: A library for ω -words, automata, and LTL," in *Automated Technology for Verification and Analysis*, ser. ATVA. Springer, 2018, pp. 543–550.
- [24] A. Church, "Application of recursive arithmetic to the problem of circuit synthesis," *Journal of Symbolic Logic*, vol. 28, no. 4, 1963.
- [25] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," *Communications of the ACM*, vol. 52, no. 11, pp. 74–84, 2009.
- [26] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 843–871, 1994.
- [27] I. Incer Romeo, A. Sangiovanni-Vincentelli, C.-W. Lin, and E. Kang, "Quotient for assume-guarantee contracts," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2018, pp. 1–11.
- [28] I. Incer, A. Badithela, J. B. Graebener, P. Mallozzi, A. Pandey, N. Rouquette, S.-J. Yu, A. Benveniste, B. Caillaud, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Pacti: Assume-guarantee contracts for efficient compositional analysis and design," *ACM Trans. Cyber-Phys. Syst.*, vol. 9, no. 1, Jan. 2025. [Online]. Available: <https://doi.org/10.1145/3704736>
- [29] T. Villa, N. Yevtushenko, R. K. Brayton, A. Mishchenko, A. Petrenko, and A. Sangiovanni-Vincentelli, *The Unknown Component Problem: Theory and Applications*. Springer US, 2012. [Online]. Available: <http://dx.doi.org/10.1007/978-0-387-68759-9>
- [30] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.
- [31] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
- [32] D. Harel, G. Katz, A. Marron, and G. Weiss, "Non-intrusive repair of reactive programs," in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. IEEE, 2012, pp. 3–12.
- [33] P. J. Guo, "Online python tutor: embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 579–584.
- [34] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 151–158.
- [35] A. Nazari, M. Amini, and M. Raghothaman, "How Does my Circuit Work?": Local explanations for the behavior of sequential circuits (Artifact)," Zenodo, 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.16884535>

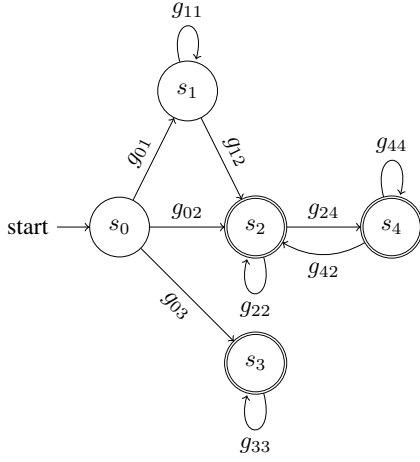


Fig. 8: Büchi automaton $M_{C,b,\phi}$ for the latch b in the circuit described in Equation 8, and with respect to the specification of Equation 7. The transition guards are provided by $g_{01} = i \wedge j \wedge \neg a \wedge b \wedge x$, $g_{02} = i \wedge j \wedge \neg a \wedge \neg b \wedge x$, $g_{03} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x)$, $g_{11} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x) \vee (\neg a \wedge b \wedge x)$, $g_{12} = i \wedge j \wedge \neg a \wedge \neg b \wedge x$, $g_{22} = a \wedge \neg b \wedge \neg x$, $g_{24} = a \wedge b \wedge \neg x$, $g_{33} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x) \vee (\neg a \wedge b \wedge x)$, $g_{42} = \neg a \wedge \neg b \wedge i \wedge j \wedge x$, and $g_{44} = (\neg a \wedge \neg i \wedge x) \vee (\neg a \wedge \neg j \wedge x) \vee (\neg a \wedge b \wedge x)$ respectively.

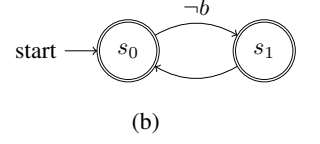
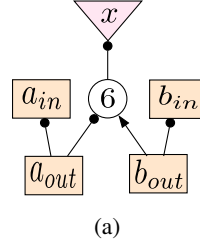


Fig. 9: (9a) Representation of the circuit from Equation 9 as an and-inverter graph. (9b) Subspec of the latch b with respect to the global specification, Gx . Observe that the output value is unconstrained in odd-indexed time steps.

APPENDIX

a) *Experimental setup:* We ran our experiments on a four-year old workstation machine with an AMD Ryzen 9 5950X CPU and 128 GB of memory running Ubuntu 21.04. We expect similar results to be obtained on most recent desktop and laptop computers.

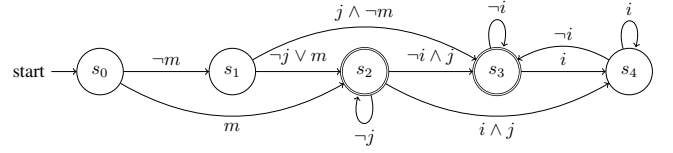


Fig. 10: Subspecification of a component m of the circuit from Equation 6 with respect to the specification in Equation III-A.

On Hyperproperty Verification, Quantifier Alternations, and Games under Partial Information

Raven Beutner 
CISPA Helmholtz Center for
Information Security
Germany

Bernd Finkbeiner 
CISPA Helmholtz Center for
Information Security
Germany

Abstract—Hyperproperties generalize traditional trace properties by relating multiple execution traces rather than reasoning about individual runs in isolation. They provide a unified way to express important requirements such as information flow and robustness properties. Temporal logics like HyperLTL capture these properties by explicitly quantifying over executions of a system. However, many practically relevant hyperproperties involve *quantifier alternations*, a feature that poses substantial challenges for automated verification. Complete verification methods require a system complementation for each quantifier alternation, making it infeasible in practice. A cheaper (but incomplete) method interprets the verification of a HyperLTL formula as a two-player game between universal and existential quantifiers. The game-based approach is significantly cheaper, facilitates interactive proofs, and allows for easy-to-check certificates of satisfaction. It is, however, limited to $\forall^*\exists^*$ properties, leaving important properties out of reach. In this paper, we show that we can use games to verify hyperproperties with arbitrary quantifier alternations by utilizing multiplayer games under *partial information*. While games under partial information are, in general, undecidable, we show that our game is played under hierarchical information and thus falls in a decidable class of games. We discuss the completeness of the game and study prophecy variables in the setting of partial information.

I. INTRODUCTION

In 2008, Clarkson and Schneider [1] coined the term *hyperproperties* for the rich class of system requirements that relate multiple executions. In contrast to *trace properties* – i.e., properties over individual executions, expressed, e.g., in linear-time temporal logics (LTL) [2] – hyperproperties can express important properties related to information flow, knowledge, and robustness. As an example, consider a system with secret input h , public input l , and public output o , and assume we want to express that the public behavior does not leak any information about the secret input. We cannot express such an information-flow requirement as a trace property in, e.g., LTL; we need to compare *multiple* executions to see if (and how) the secret input impacts the output. Instead, we can express it as a hyperproperty in HyperLTL [3], an extension of LTL with explicit quantification over execution traces. For example,

$$\forall \pi_1. \forall \pi_2. \Box(l_{\pi_1} \leftrightarrow l_{\pi_2}) \rightarrow \Box(o_{\pi_1} \leftrightarrow o_{\pi_2}). \quad (\text{OD})$$

requires that any pair of executions π_1, π_2 with identical public input also has the same output, i.e., the output is fully determined by the public input and thus cannot possibly leak the secret input (cf. *observational determinism* [4]).

While originating in the study of information flow, hyperproperties have since been established as a much more general framework that captures properties from many different areas, including, e.g., knowledge properties in multi-agent systems (MAS) [5], [6]. As an example, consider some MAS with agents $1, \dots, n$, and some LTL property ψ , and assume that we want to verify that there exists at least one execution of the MAS such that agent i knows that ψ holds (e.g., some adversary knowing some secret). Formally, *knowing* that ψ holds on some execution trace π_1 means that ψ must hold on all traces π_2 that are indistinguishable from π_1 for agent i (cf. [7]), which is a hyperproperty. We can easily express this property in HyperLTL, as follows

$$\exists \pi_1. \forall \pi_2. \pi_1 \equiv_i \pi_2 \rightarrow \psi[\pi_2], \quad (\text{K}_1)$$

where we write $\psi[\pi_2]$ to indicate that ψ holds on trace π_2 , and $\pi_1 \equiv_i \pi_2$ denotes that executions π_1 and π_2 appear indistinguishable under agent i 's observations. Using the flexibility of quantification, we can also express nested knowledge properties. For example, we can express that, on some execution, agent i knows that agent j does *not* know whether ψ holds:

$$\begin{aligned} \exists \pi_1. \forall \pi_2. \exists \pi_3. \exists \pi_4. \pi_1 \equiv_i \pi_2 \rightarrow \\ (\pi_2 \equiv_j \pi_3 \wedge \pi_2 \not\equiv_j \pi_4 \wedge \psi[\pi_3] \wedge \neg \psi[\pi_4]), \end{aligned} \quad (\text{K}_2)$$

i.e., for every trace π_2 that agent i cannot distinguish from π_1 , there exist two traces π_3, π_4 that agent j cannot distinguish from π_2 , one of which satisfies ψ and one violates ψ .

Verification of HyperLTL: In this paper, we study the model-checking problem for HyperLTL (or, more generally, of hyperproperties that can be expressed using HyperLTL-style quantification over system paths or traces [8]–[10]). Unsurprisingly, the quantifier prefix of the HyperLTL formula directly impacts the complexity of this verification problem. For alternation-free formulas (e.g., OD), verification can be reduced to the verification of an LTL property on the self-composition of the system [11], [12], which is very efficient. Verification gets much more challenging when the formula includes quantifier alternations as used in K_1 , K_2 , and many other HyperLTL formulas studied in the literature [13]–[15]. Complete approaches require one system complementation for each alternation in the formula [3], [12], making it infeasible in practice.

Game-based Verification: A cheaper (but incomplete) verification method for $\forall^*\exists^*$ formulas (i.e., formulas where an arbitrary number of universal quantifiers is followed by an arbitrary number of existential quantifiers) is based on a strategy-based interpretation of existential quantification [16], [17]. The key idea is to interpret the verification of a HyperLTL formula of the form $\forall\pi_1.\exists\pi_2.\psi$ (where ψ is the LTL body) as a *game* between two players. A refuter controls the universally quantified trace by moving through a copy of the underlying system, thereby constructing a concrete trace for π_1 . The verifier reacts to the moves by the refuter and moves through a separate copy of the system, thereby producing a concrete trace for π_2 . The goal of the verifier is to ensure that π_1 and π_2 , together, satisfy ψ . We can think of the verifier’s strategy as providing a step-wise construction of a concrete witness trace for π_2 (akin to a Skolem function). This game-based approach is sound (i.e., if the verifier wins, the hyperproperty is satisfied by the system) and computationally cheaper than complementation-based approaches. Moreover, the game-based approach also allows for interactive proofs and witnesses of satisfaction. For example, we can use the game-based framework to let the user construct a strategy *interactively* [18], allowing verification even in situations where automated techniques do not scale. Likewise, we can use a winning strategy for the verifier as an (easy-to-check) certificate that the property is satisfied [19].

Unsoundness Beyond $\forall^*\exists^*$: The game-based verification approach of Coenen et al. [16] and its descendants have proven themselves in many situations (cf. Section VII). However, since its inception, the approach has been limited to $\forall^*\exists^*$ properties. Intuitively, as soon as we consider properties beyond $\forall^*\exists^*$, the step-wise selection of the traces leads to unsoundness, i.e., cases where a winning strategy exists even though the property is violated; we give a concrete instance in Example 1. Consequently, for properties outside the $\forall^*\exists^*$ fragment, no effective verification approximation exists (cf. Section VII), nor does there exist any approach that allows interactive proofs or satisfaction certificates.

Partial Information: In this paper, we present a novel game-based method that allows us to soundly verify *arbitrary* quantifier structures. Our key contribution is the observation that we need to reason about *partial information*. In our game-based encoding, we consider multiple players, each of whom controls a unique trace in the HyperLTL formula. We then carefully design an observation model for each player to ensure soundness. Intuitively, our observations ensure that each player controlling some trace π can only observe the state sequence from traces that are quantified *before* π . We thus obtain a multiplayer game played under partial information that, if won by a certain group of players, ensures that the formula holds on the given system (Section IV).

Hierarchical Information: In general, multiplayer games under incomplete information are undecidable. We show that our verification game falls in a well-known class of games that can be solved effectively. Namely, games where the information of the players is *hierarchical* [20]–[26].

Completeness and Prophecy Variables: Similar to the $\forall^*\exists^*$ game [16], [17], our game-based approach using partial information is incomplete: In some cases, the property holds, but no winning strategy exists. While incomplete for $\forall^*\exists^*$ properties, we show that our approach is complete for $\exists^*\forall^*$ properties (Section V). This gives rise to a sound-and-complete verification method for all properties with at most one quantifier alternation (via complementation), at the cost of introducing partial information. More generally, we study the use of prophecy variables [17], [27] in our verification game, allowing a player to peek at the future temporal behavior of other players (Section VI).

Applications: The core contribution that partial information enables game-based verification of arbitrary hyperproperties creates a plethora of new possibilities for hyperproperty verification: Our game-based view (1) leverages techniques for solving partial information games (cf. Section VII) for automated verification; (2) facilitates interactive proof of hyperproperties beyond $\forall^*\exists^*$ by letting the user construct strategies; (3) enables strategies as easy-to-check certificates for satisfaction; and (4) supports prophecies to soundly strengthen the approach, both in automated and interactive verification.

Full Version: Proofs of all results can be found in the full version [28].

II. PRELIMINARIES

Kripke Structures: As the basic system model, we use finite-state Kripke structures. We assume that AP is a fixed set of *atomic propositions* (AP). A Kripke structure (KS) is a tuple $\mathcal{K} = (S, s_{init}, \mathbb{D}, \kappa, \ell)$, where S is a finite set of states, $s_{init} \notin S$ is a dedicated initial state (*not* part of S), $\mathbb{D}$ is a finite set of directions, $\kappa : (S \uplus \{s_{init}\}) \times \mathbb{D} \rightarrow S$ is the transition function, and $\ell : (S \uplus \{s_{init}\}) \rightarrow 2^{AP}$ labels each state with an evaluation of the APs.¹ A path in $\mathcal{K}$ is an infinite sequence $\tau \in (S \uplus \{s_{init}\})^\omega$ such that $\tau(0) = s_{init}$, and for every $i \in \mathbb{N}$, there exists some $d \in \mathbb{D}$ such that $\tau(i+1) = \kappa(\tau(i), d)$. We define $Paths(\mathcal{K})$ as the set of all paths in $\mathcal{K}$. Each path τ denotes an associated trace $\ell(\tau) := \ell(\tau(0))\ell(\tau(1))\ell(\tau(2)) \dots \in (2^{AP})^\omega$ defined by applying ℓ pointwise. We define $Traces(\mathcal{K}) := \{\ell(\tau) \mid \tau \in Paths(\mathcal{K})\} \subseteq (2^{AP})^\omega$ as the set of all traces generated by $\mathcal{K}$.

Linear-Time Temporal Logic: Linear-time temporal logic (LTL) [2] formulas are defined as follows

$$\psi := a \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi \mathcal{U} \psi,$$

where $a \in AP$ is an atomic proposition. The basic formula a requires that the AP a holds in the current state, $\bigcirc\psi$ requires that ψ holds in the *next* step, and $\psi_1 \mathcal{U} \psi_2$ requires that ψ_1 holds *until* ψ_2 eventually holds. We use the usual derived constants and connectives *true*, *false*, $\vee$, $\rightarrow$, $\leftrightarrow$, and

¹We use slightly unconventional notation in two places: Firstly, we assume a dedicated initial state s_{init} , simplifying the addition of prophecies. Secondly, we use *directions* to uniquely identify successor states, simplifying our game construction. Traditionally, transition functions in Kripke structures are functions $S \rightarrow 2^S \setminus \{\emptyset\}$ that map each state to a non-empty set of successor states. We can easily transform such a transition function into a directed function $S \times \mathbb{D} \rightarrow S$ by using sufficiently many directions.

the temporal operators *eventually* $\Diamond\psi := \text{true}\mathcal{U}\psi$, and *globally* $\Box\psi := \neg\Diamond\neg\psi$. Given a trace $t \in (2^{AP})^\omega$, we define the semantics of LTL for each time point $i \in \mathbb{N}$ as follows:

$$\begin{aligned} t, i \models a & \quad \text{iff} \quad a \in t(i) \\ t, i \models \psi_1 \wedge \psi_2 & \quad \text{iff} \quad t, i \models \psi_1 \text{ and } t, i \models \psi_2 \\ t, i \models \neg\psi & \quad \text{iff} \quad t, i \not\models \psi \\ t, i \models \bigcirc\psi & \quad \text{iff} \quad t, i+1 \models \psi \\ t, i \models \psi_1 \mathcal{U} \psi_2 & \quad \text{iff} \quad \exists k \geq i. t, k \models \psi_2 \text{ and} \\ & \quad \forall i \leq j < k. t, j \models \psi_1 \end{aligned}$$

We write $t \models_{LTL} \psi$ if t satisfies ψ , i.e., $t, 0 \models \psi$.

Deterministic Parity Automata: A deterministic parity automaton (DPA) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, c)$ where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, and $c : Q \rightarrow \mathbb{N}$ colors each state with a natural number. For an infinite word $u \in \Sigma^\omega$, we define $\text{run}(\mathcal{A}, u) \in Q^\omega$ as the unique run of $\mathcal{A}$ on u . Formally, $\text{run}(\mathcal{A}, u)$ is the unique word in Q^ω such that $\text{run}(\mathcal{A}, u)(0) = q_0$ and for every $i \in \mathbb{N}$, $\text{run}(\mathcal{A}, u)(i+1) = \delta(\text{run}(\mathcal{A}, u)(i), u(i))$. That is, we start the run in the initial state q_0 and progress by following $\mathcal{A}$'s transition function using the letters from u . The acceptance condition in DPAs is based on the color of the states (as given by c). An infinite run in Q^ω is accepting if the minimal color that occurs infinitely often is even. We write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ for the language of the automaton, which consists of all words where the unique run is accepting. In this paper, we use a parity acceptance condition as DPAs capture every ω -regular property and thus every LTL-expressible property:

Lemma 1 ([29], [30]). *For every LTL formula ψ , we can effectively construct a DPA $\mathcal{A}_\psi = (2^{AP}, Q_\psi, q_0, \delta_\psi, c_\psi)$ such that $\mathcal{L}(\mathcal{A}_\psi) = \{t \in (2^{AP})^\omega \mid t \models_{LTL} \psi\}$.*

We emphasize that our construction also applies to other automaton types. For example, if the LTL body of our hyperproperty can be expressed as a *deterministic* Büchi (resp. safety) automaton, our later construction in Section IV yields a Büchi (resp. safety) game.

HyperLTL: HyperLTL [3] extends LTL with explicit quantification over (execution) traces of the system. Let $\mathcal{V} = \{\pi_1, \pi_2, \dots\}$ be a set of *trace variables*. HyperLTL formulas are generated by the following grammar

$$\begin{aligned} \psi &:= a_\pi \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi \mathcal{U} \psi \\ \varphi &:= \forall\pi. \varphi \mid \exists\pi. \varphi \mid \psi \end{aligned}$$

where $a \in AP$ is an atomic proposition, and $\pi \in \mathcal{V}$ is a trace variable. Each HyperLTL formula thus has the form $\varphi = Q_1\pi_1 \dots Q_n\pi_n. \psi$, where $Q_1, \dots, Q_n \in \{\forall, \exists\}$ are quantifiers, $\pi_1, \dots, \pi_n \in \mathcal{V}$ are trace variables, and ψ is an LTL formula over trace-variable-indexed APs. The formula quantifies over traces $\pi_1, \dots, \pi_n$ in the system (in typical first-order semantics) and requires that the resulting combination of n traces satisfies the temporal requirement expressed by the trace-variable-indexed LTL formula ψ .

A trace assignment is a partial function $\Pi : \mathcal{V} \rightarrow (2^{AP})^\omega$ that maps trace variables to traces. Given Π we can evaluate the LTL body ψ in each time point $i \in \mathbb{N}$:

$$\begin{aligned} \Pi, i \models a_\pi & \quad \text{iff} \quad a \in \Pi(\pi)(i) \\ \Pi, i \models \psi_1 \wedge \psi_2 & \quad \text{iff} \quad \Pi, i \models \psi_1 \text{ and } \Pi, i \models \psi_2 \\ \Pi, i \models \neg\psi & \quad \text{iff} \quad \Pi, i \not\models \psi \\ \Pi, i \models \bigcirc\psi & \quad \text{iff} \quad \Pi, i+1 \models \psi \\ \Pi, i \models \psi_1 \mathcal{U} \psi_2 & \quad \text{iff} \quad \exists k \geq i. \Pi, k \models \psi_2 \text{ and} \\ & \quad \forall i \leq j < k. \Pi, j \models \psi_1 \end{aligned}$$

Temporal and Boolean operators are evaluated as for LTL. Whenever we evaluate an indexed AP a_π , we look at the trace bound to π and check if a currently holds on this trace. Given a KS $\mathcal{K}$, the quantifier prefix in HyperLTL then adds traces to the trace assignment in typical first-order fashion:

$$\begin{aligned} \Pi \models_{\mathcal{K}} \psi & \quad \text{iff} \quad \Pi, 0 \models \psi \\ \Pi \models_{\mathcal{K}} \forall\pi. \varphi & \quad \text{iff} \quad \forall t \in \text{Traces}(\mathcal{K}). \Pi[\pi \mapsto t] \models_{\mathcal{K}} \varphi \\ \Pi \models_{\mathcal{K}} \exists\pi. \varphi & \quad \text{iff} \quad \exists t \in \text{Traces}(\mathcal{K}). \Pi[\pi \mapsto t] \models_{\mathcal{K}} \varphi \end{aligned}$$

We say $\mathcal{K}$ satisfies φ , written $\mathcal{K} \models \varphi$, if $\emptyset \models_{\mathcal{K}} \varphi$, where $\emptyset$ denotes the trace assignment with empty domain.

III. GAME-BASED VERIFICATION OF $\forall^*\exists^*$

Model-checking a HyperLTL formula with k quantifier alternations is k -fold exponential [3], [8], and complete methods typically utilize expensive operations like automata complementation [12] and inclusion checking [31]. For $\forall^*\exists^*$ properties, we can soundly (but incompletely) *approximate* the expensive model-checking problem by, instead, constructing a game and searching for a *strategy* that defines witness paths for existentially quantified traces [16], [17].

A. Parity Games

To model the dynamics of the verification game, we use a turn-based game played between a verifier and a refuter (as done by [17]).

A parity game (PG) is a tuple $\mathcal{G} = (V_{\mathfrak{V}}, V_{\mathfrak{R}}, v_{\text{init}}, \mathbb{D}, E, c)$, where $V := V_{\mathfrak{V}} \uplus V_{\mathfrak{R}}$ is the set of game vertices, partitioned into vertices controlled by the verifier ($V_{\mathfrak{V}}$) and refuter ($V_{\mathfrak{R}}$), $v_{\text{init}} \in V$ is the initial vertex of the game, $\mathbb{D}$ is a set of directions, $E : V \times \mathbb{D} \rightarrow V$ is the transition function of the game, and $c : V \rightarrow \mathbb{N}$ assigns each state a color used for the parity acceptance condition.

The game is played on an underlying graph whose vertices are controlled by the verifier ($\mathfrak{V}$) or refuter ($\mathfrak{R}$). Whenever the game is in a vertex controlled by a player $p \in \{\mathfrak{V}, \mathfrak{R}\}$, the respective player can determine to which vertex the game should progress by choosing some direction from $\mathbb{D}$. A *strategy* for the verifier is a function $\sigma : V^* \cdot V_{\mathfrak{V}} \rightarrow \mathbb{D}$. The strategy reads a sequence of vertices $v_1 \dots v_n$ (ending in a vertex $v_n \in V_{\mathfrak{V}}$ controlled by the verifier) and determines a direction $\sigma(v_1 \dots v_n) \in \mathbb{D}$ in which the game should progress. A play $\rho \in V^\omega$ is compatible with a strategy σ for $\mathfrak{V}$ if $\rho(0) = v_{\text{init}}$ (i.e., the play starts in $\mathcal{G}$'s initial vertex), and for every $i \in \mathbb{N}$,

with $\rho(i) \in V_{\mathfrak{V}}$, we have $\rho(i+1) = E(\rho(i), \sigma(\rho[0, i]))$. That is, we construct the play iteratively; Whenever the game is in a vertex controlled by the verifier, we query strategy σ on the current prefix to obtain a direction and update the vertex based on $\mathcal{G}$'s transition function. We say a play $\rho \in V^\omega$ is *even* if the minimal color that appears infinitely often on ρ (according to coloring c) is even (similar to the acceptance condition used for DPAs). The verifier wins $\mathcal{G}$ if there exists a strategy σ for the verifier such that every play compatible with σ is even.

B. The Verification Game for $\forall^*\exists^*$

Assume a fixed system $\mathcal{K} = (S, s_{init}, \mathbb{D}, \kappa, \ell)$ and a $\forall\exists$ HyperLTL formula $\forall\pi_1.\exists\pi_2.\psi$. For our game construction, we represent the temporal requirement expressed by ψ – the LTL body of φ – as a deterministic automaton. Recall that the atomic propositions in ψ are indexed with trace variables, i.e., ψ is an LTL formula over

$$AP_\psi := \{a_\pi \mid a \in AP, \pi \in \{\pi_1, \pi_2\}\}.$$

We assume that $\mathcal{A}_\psi = (2^{AP_\psi}, Q_\psi, q_{0,\psi}, \delta_\psi, c_\psi)$ is a DPA over alphabet 2^{AP_ψ} that recognizes ψ , i.e., $\mathcal{L}(\mathcal{A}_\psi) = \{t \in (2^{AP_\psi})^\omega \mid t \models_{LTL} \psi\}$ (cf. Lemma 1). We can then define a parity game, denoted $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$, that captures the iterative construction of traces [17]:

Definition 1 ($\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$, [17]). Define the parity game $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$ by $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists} := (V_{\mathfrak{V}}, V_{\mathfrak{R}}, v_{init}, \mathbb{D}, E, c)$, where

- $V_{\mathfrak{V}} := \{\langle s_1, s_2, q, \mathfrak{V} \rangle \mid s_1, s_2 \in S \uplus \{s_{init}\} \wedge q \in Q_\psi\}$,
- $V_{\mathfrak{R}} := \{\langle s_1, s_2, q, \mathfrak{R} \rangle \mid s_1, s_2 \in S \uplus \{s_{init}\} \wedge q \in Q_\psi\}$,
- $v_{init} := \langle s_{init}, s_{init}, q_{0,\psi}, \mathfrak{R} \rangle$,
- the set of direction $\mathbb{D}$ is the same as in $\mathcal{K}$,
- the transition function $E : V \times \mathbb{D} \rightarrow V$ is defined by

$$E(\langle s_1, s_2, q, \mathfrak{V} \rangle, d) := \langle s_1, \kappa(s_2, d), q, \mathfrak{R} \rangle$$

$$E(\langle s_1, s_2, q, \mathfrak{R} \rangle, d) :=$$

$$\left\langle \kappa(s_1, d), s_2, \delta_\psi\left(q, \bigcup_{i=1}^2 \{a_{\pi_i} \mid a \in \ell(s_i)\}\right), \mathfrak{V} \right\rangle,$$

- $c(\langle s_1, s_2, q, p \rangle) := c_\psi(q)$.

In our game, each vertex $\langle s_1, s_2, q, p \rangle$ tracks a state s_1 for π_1 (called the π_1 -copy), a state s_2 for π_2 (called the π_2 -copy), the current state q of $\mathcal{A}_\psi$, and the current player $p \in \{\mathfrak{V}, \mathfrak{R}\}$. In the initial vertex v_{init} , every system copy starts in the initial state, and $\mathcal{A}_\psi$ begins tracking in its initial state $q_{0,\psi}$. Intuitively, in each round of the game, the refuter can update the π_1 -copy by moving along some transition in $\mathcal{K}$, followed by the verifier updating the π_2 -copy; afterward, the game repeats. Formally, whenever in a vertex $\langle s_1, s_2, q, \mathfrak{V} \rangle$, the verifier can choose a direction $d \in \mathbb{D}$, and the π_2 -copy is updated to $\kappa(s_2, d)$ (the first case in the definition of E). Analogously, when in vertex $\langle s_1, s_2, q, \mathfrak{R} \rangle$ the refuter can update the π_1 -copy along some direction (the second case in the definition of E). When the refuter moves a round of the game has concluded, so we update the state of $\mathcal{A}_\psi$. For each $i \in \{1, 2\}$, we thus read of the APs that currently hold in state

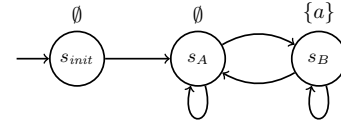


Fig. 1: Simple Kripke structure over $AP = \{a\}$

s_i ($\ell(s_i) \subseteq AP$) and index all these APs with π_i to obtain a letter $\bigcup_{i=1}^2 \{a_{\pi_i} \mid a \in \ell(s_i)\} \in 2^{AP_\psi}$, which we feed to $\mathcal{A}_\psi$'s transition function.

As we track separate states for π_1 and π_2 , every infinite play in $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$ defines two paths in $\mathcal{K}$; one for π_1 (where each step is controlled by the refuter), and one for π_2 (controlled by the verifier). In $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$, each vertex $\langle s_1, s_2, q, p \rangle$ is assigned color $c_\psi(q)$ using $\mathcal{A}_\psi$'s coloring function, so an infinite play in $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$ is won by the verifier iff the paths constructed for π_1, π_2 during the gameplay are accepted by $\mathcal{A}_\psi$ and thus satisfy ψ . Any winning strategy for $\mathfrak{V}$ thus step-wise constructs a witness trace for π_2 , no matter how the refuter constructs π_1 . It is not hard to see that the existence of a winning strategy for the verifier thus implies that we can always find a witness trace for π_2 in the HyperLTL semantics:

Lemma 2 ([17]). If the verifier wins $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$, then $\mathcal{K} \models \varphi$.

Remark 1. The game-based approach can be used for automated verification by constructing the game and solving it via an off-the-shelf parity solver (the original motivation of [16], [17]). However, the appeal of Lemma 2 is much broader. For example, the user can construct a strategy by using domain knowledge, which enables interactive verification even in situations where automated model-checking does not scale (see, e.g., [18]). Likewise, checking if a given strategy for the verifier wins $\mathcal{G}_{\mathcal{K},\varphi}^{\forall\exists}$ is often easier than computing a strategy from scratch; strategies are easy-to-check certificates.

IV. GAME-BASED VERIFICATION BEYOND $\forall^*\exists^*$

The game-based approach from the previous section soundly approximates the semantics of $\forall^*\exists^*$ formulas. Unfortunately, the approach is limited to $\forall^*\exists^*$ properties and becomes unsound when considering properties beyond $\forall^*\exists^*$.

Example 1. Consider the Kripke structure $\mathcal{K}$ over $AP = \{a\}$ in Figure 1 and the $\exists^1\forall^1$ HyperLTL formula

$$\varphi := \exists\pi_1.\forall\pi_2.(\bigcirc\bigcirc\bigcirc a_{\pi_1}) \leftrightarrow (\bigcirc\bigcirc a_{\pi_2}),$$

where the LTL body expresses that AP a should hold in the third step on π_1 iff it holds in the second step on π_2 . Clearly, $\mathcal{K} \not\models \varphi$; no matter what fixed trace we choose for π_1 , we can always find a trace for π_2 that violates the LTL body.

Now let us naïvely adopt the game used in Section III to this $\exists\forall$ property. That is, we, again, maintain states for all trace variables and let the verifier and refuter iteratively update the existentially and universally quantified system copies, respectively. The resulting game is won by the verifier: During the gameplay, the refuter (who controls the state for the universally quantified π_2) has to decide in the second step

whether or not AP a should hold (by moving to s_B or s_A). Only later in the game (in the third round) does the verifier choose if a holds on π_1 . By that time, the verifier can thus react to what the refuter has done in the previous step and set a on π_1 appropriately, ensuring that $(\bigcirc\bigcirc\bigcirc a_{\pi_1}) \leftrightarrow (\bigcirc\bigcirc a_{\pi_2})$ is satisfied. The game is won by the verifier, even though the property does not hold.

A. Multiplayer Games and Partial Information

In this paper, we propose a novel game-based approximation that applies to arbitrary quantifier structures. Our simple yet powerful observation is that the unsoundness is directly linked to the *knowledge* of the player. In Example 1, the verifier could win the game as it can observe what the refuter did in the previous steps. In contrast, in the semantics of an $\exists\forall$ property, any witness for the existential quantifier must be *independent* of the choice for the universal quantifier. By identifying knowledge as the core reason for unsoundness, we can extend the game-based approach to *arbitrary* HyperLTL formulas by utilizing *partial information*. The technical challenge is then to link the knowledge/information of a player to the first-order semantics of HyperLTL.

As our underlying game formalism, we use (sequential) multiplayer parity games [32], which extend parity games with multiple players and partial information. A multiplayer parity game under incomplete information (MPG_{ii}) is a tuple

$$\mathcal{G} = (\mathbb{P}, \{V_p\}_{p \in \mathbb{P}}, v_{init}, \mathbb{D}, E, \{\sim_p\}_{p \in \mathbb{P}}, c),$$

where $\mathbb{P}$ is a finite set of players; for each $p \in \mathbb{P}$, V_p is a finite set of vertices controlled by p . We write $V := \bigsqcup_{p \in \mathbb{P}} V_p$ for the set of all vertices in the game (which we assume to be disjoint). $v_{init} \in V$ is the initial vertex of the game, $\mathbb{D}$ is a set of directions, and $E : V \times \mathbb{D} \rightarrow V$ is the transition function of the game. For each player $p \in \mathbb{P}$, $\sim_p \subseteq V \times V$ is an equivalence relation on the set of vertices. Lastly, $c : V \rightarrow \mathbb{N}$ assigns each vertex a color.

An MPG_{ii} models the joint behavior of the players in $\mathbb{P}$, where each player $p \in \mathbb{P}$ controls a disjoint set of vertices V_p . If the game is currently in a vertex in V_p , player p can determine where the game should move next by choosing some direction from $\mathbb{D}$. We obtain a standard two-player game (cf. Section III) by setting $\mathbb{P} = \{\mathfrak{V}, \mathfrak{R}\}$. Moreover, each player $p \in \mathbb{P}$ is assigned an indistinguishability relation $\sim_p$, i.e., if $v \sim_p v'$, player p cannot distinguish between v and v' . As usual, we assume that each player is at least able to distinguish whether or not it controls the current vertex. That is, for every $v \sim_p v'$, we either have $v, v' \in V_p$ or $v, v' \in V \setminus V_p$.

Strategies and Plays: Two finite plays $\rho_1, \rho_2 \in V^*$ are indistinguishable for player $p \in \mathbb{P}$, written $\rho_1 \sim_p \rho_2$, if $|\rho_1| = |\rho_2|$ and for every $0 \leq i < |\rho_1|$, $\rho_1(i) \sim_p \rho_2(i)$. A strategy for player $p \in \mathbb{P}$ is a function $\sigma_p : V^* \cdot V_p \rightarrow \mathbb{D}$, such that $\sigma_p(\rho_1) = \sigma_p(\rho_2)$ whenever $\rho_1 \sim_p \rho_2$. The strategy reads a sequence of vertices $v_1 \cdots v_n$ (ending in a vertex $v_n \in V_p$) and determines a direction $\sigma_p(v_1 \cdots v_n) \in \mathbb{D}$ in which the game should progress. This decision must conform to the agent's observations, i.e., if two finite plays appear indistinguishable

for p , strategy σ_p must choose the same direction on both plays. Within the game, we obtain infinite plays in $\mathcal{G}$ by letting strategies for all players interact. A *global strategy* $\{\sigma_p\}_{p \in \mathbb{P}}$ assigns each player $p \in \mathbb{P}$ a strategy σ_p . Every global strategy $\{\sigma_p\}_{p \in \mathbb{P}}$ defines a unique infinite play $\rho \in V^\omega$, where $\rho(0) = v_{init}$ (i.e., the play starts in $\mathcal{G}$'s initial vertex), and for every $i \in \mathbb{N}$, we have $\rho(i+1) = E(\rho(i), \sigma_p(\rho[0, i]))$ where $p \in \mathbb{P}$ is the unique player with $\rho(i) \in V_p$. That is, whenever the game is currently in a vertex controlled by player p , we query p 's strategy on the current prefix $\rho[0, i]$ to obtain a direction and update the game's vertex along that direction. As before, we say a play ρ is *even* if the minimal color that appears infinitely often on ρ is even. We are interested in checking if a given *coalition* of players $A \subseteq \mathbb{P}$ can win the game. A coalition $A \subseteq \mathbb{P}$ wins $\mathcal{G}$, written $wins(A, \mathcal{G})$, if there exists strategies $\{\sigma_p\}_{p \in A}$ for the players in A , such that, no matter what strategies other players use, i.e., for every possible $\{\sigma_p\}_{p \in \mathbb{P} \setminus A}$, the play resulting from the combined global strategy $\{\sigma_p\}_{p \in \mathbb{P}}$ is even. That is, if the players in A follow their strategies in $\{\sigma_p\}_{p \in A}$, the resulting play satisfies the parity winning condition no matter how the other players behave.

B. HyperLTL Verification as an MPG_{ii}

We now present the core contribution of this paper: MPG_{ii}s allow for the sound verification of *arbitrary* HyperLTL formulas. For this, assume

$$\varphi = Q_1 \pi_1 \dots Q_n \pi_n \cdot \psi$$

is a fixed HyperLTL formula over trace variables $\pi_1, \dots, \pi_n$ with *arbitrary* quantifier structure. As in Section III, we assume that $\mathcal{A}_\psi = (2^{AP_\psi}, Q_\psi, q_{0,\psi}, \delta_\psi, c_\psi)$ is a DPA over alphabet $AP_\psi := \{a_\pi \mid a \in AP, \pi \in \{\pi_1, \dots, \pi_n\}\}$ that recognizes ψ (cf. Lemma 1).

Definition 2 ($\mathcal{G}_{\mathcal{K}, \varphi}$). Define the MPG_{ii} $\mathcal{G}_{\mathcal{K}, \varphi}$ by

$$\mathcal{G}_{\mathcal{K}, \varphi} := (\mathbb{P}, \{V_p\}_{p \in \mathbb{P}}, v_{init}, \mathbb{D}, E, \{\sim_p\}_{p \in \mathbb{P}}, c),$$

where

- $\mathbb{P} := \{1, \dots, n\}$,
- for each $p \in \mathbb{P}$,

$$V_p := \{ \langle s_1, \dots, s_n, q, p \rangle \mid s_1, \dots, s_n \in S \uplus \{s_{init}\}, q \in Q_\psi \},$$

- $v_{init} := \langle s_{init}, \dots, s_{init}, q_{0,\psi}, 1 \rangle$,
- the set of direction $\mathbb{D}$ is the same as in $\mathcal{K}$,
- the transition function $E : V \times \mathbb{D} \rightarrow V$ is defined by

$$E(\langle s_1, \dots, s_n, q, p \rangle, d) := \langle s_1, \dots, s_{p-1}, \kappa(s_p, d), s_{p+1}, \dots, s_n, q, \text{next}(p) \rangle,$$

for $p > 1$ and for $p = 1$ define

$$E(\langle s_1, \dots, s_n, q, 1 \rangle, d) := \langle \kappa(s_1, d), s_2, \dots, s_n, \delta_\psi(q, \bigcup_{i=1}^n \{a_{\pi_i} \mid a \in \ell(s_i)\}), \text{next}(1) \rangle,$$

- where $\text{next}(p) := p + 1$ if $p < n$ and $\text{next}(n) := 1$,
- for each $p \in \mathbb{P}$,

$$\sim_p := \left\{ \left(\langle s_1, \dots, s_n, q, p' \rangle, \langle s'_1, \dots, s'_n, q', p'' \rangle \right) \mid p' = p'' \wedge \forall 1 \leq j \leq p. s_j = s'_j \right\},$$

- $c(\langle s_1, \dots, s_n, q, p \rangle) := c_\psi(q)$.

The first key idea is to represent each trace variable in the formula as a separate player, so $\mathbb{P} = \{1, \dots, n\}$. The vertices in $\mathcal{G}_{\mathcal{K}, \varphi}$ are of the form $\langle s_1, \dots, s_n, q, p \rangle$, where $s_1, \dots, s_n$ track the current state of the system copies for $\pi_1, \dots, \pi_n$, respectively, $q \in Q_\psi$ tracks ψ , and $p \in \mathbb{P}$ determines which player controls this vertex. Each infinite play in $\mathcal{G}_{\mathcal{K}, \varphi}$, therefore, defines n concrete paths (and thus traces) for $\pi_1, \dots, \pi_n$. As expected, we start each system in $\mathcal{K}$'s initial state s_{init} , start $\mathcal{A}_\psi$ in $q_{0, \psi}$, and let player 1 begin. Similar to Section III, a vertex $\langle s_1, \dots, s_n, q, p \rangle$ is assigned color $c_\psi(q)$. The transition function then allows the players to update their system copy. Whenever in a vertex $\langle s_1, \dots, s_n, q, p \rangle$ where $p > 1$ (the first case in the definition of E), the direction d (which is chosen by player p controlling this vertex) updates the π_p -copy by moving along the chosen direction $d \in \mathbb{D}$ to $\kappa(s_p, d)$. Afterward, it is the next player's turn: $\text{next}(p)$ increases p by 1 and cycles back to player 1 once the last player (player n) has acted. The players thus take turns updating their system state; first, player 1 updates the state of the π_1 -copy, then player 2 updates the state of the π_2 -copy, and so forth, until, finally, player n updates the state of the π_n -copy, and the game repeats with player 1. When it is player 1's turn (the second case in the definition of E), a game round has just concluded. In this case, the direction chosen by player 1 is used to update the π_1 copy (similar to the other rounds). At the same time, we update the automaton state of $\mathcal{A}_\psi$ by reading the AP evaluation of $s_1, \dots, s_n$, obtaining a letter $\bigcup_{i=1}^n \{a_{\pi_i} \mid a \in \ell(s_i)\} \in 2^{AP_\psi}$.

The last key ingredient is the partial information of each player. The core problem of the game from Section III was that the players could observe the global state of the game, leading to unsound behavior. MPG_{ii} s allow us to precisely determine the information that each player can act on. Once we have observed that knowledge is the key to a sound verification game, we can align the player's information with the HyperLTL semantics: In a HyperLTL formula $\mathbb{Q}\pi_1 \dots \mathbb{Q}\pi_{i-1}. \exists \pi_i. \mathbb{Q}\pi_{i+1} \dots \mathbb{Q}\pi_n. \psi$, the choice for π_i is only based on the traces $\pi_1, \dots, \pi_{i-1}$ as those are the traces that are already added to the trace assignment in the semantics of HyperLTL. We can directly express this in our game definition: For a player p (that controls π_p), two vertices $\langle s_1, \dots, s_n, q, p' \rangle$ and $\langle s'_1, \dots, s'_n, q', p'' \rangle$ appear indistinguishable if it is the same player's turn ($p' = p''$) and $s_j = s'_j$ for all $j \leq p$, i.e., the state of all traces quantified before π_p agrees.

C. Soundness

To use our game as a verification method, we are interested in the strategic ability of all players controlling existentially

quantified system copies. That is, we define

$$\mathbb{P}_\exists := \{i \in \mathbb{P} \mid \mathbb{Q}_i = \exists\}.$$

We can then show that our game under partial information constitutes a sound verification approach:

Theorem 1. *If $\text{wins}(\mathbb{P}_\exists, \mathcal{G}_{\mathcal{K}, \varphi})$, then $\mathcal{K} \models \varphi$.*

Example 2. *Consider some formula $\forall \pi_1. \exists \pi_2. \forall \pi_3. \psi$. In our game definition, the player controlling π_2 can observe the current state of the π_1 -copy and thus react to the behavior of player 1. However, it cannot base its decision on the behavior of player 3. In the special case of $\exists \pi_1. \forall \pi_2$ properties (like in Example 1), player 1 can only observe its own state. It must thus use the same sequence of directions (and thus define the same witness paths), no matter how player 2 behaves (cf. [33]). In Example 1, player 1 would thus lose this game; there is no winning behavior in the third step without observing player 2's behavior in the second step.*

It is not hard to see that in the special case of $\forall^* \exists^*$ properties, the MPG_{ii} $\mathcal{G}_{\mathcal{K}, \varphi}$ coincides with the game $\mathcal{G}_{\mathcal{K}, \varphi}^{\forall \exists}$ from Section III.

Lemma 3. *Let $\varphi = \forall \pi_1. \exists \pi_2. \psi$. Then $\text{wins}(\{2\}, \mathcal{G}_{\mathcal{K}, \varphi})$ if and only if the verifier wins $\mathcal{G}_{\mathcal{K}, \varphi}^{\forall \exists}$ (cf. Section III).*

Similar to the $\forall^* \exists^*$ game from Section III, we can use $\mathcal{G}_{\mathcal{K}, \varphi}$ not only for automated verification but also as a foundation for interactive verification and certificates (cf. Remark 1).

D. Hierarchical Information

Multiplayer games under imperfect information are often undecidable [20], [34], i.e., there exists no general algorithm to check if a given group of players can win the game. Fortunately, there exists a well-known class of imperfect information games that we can solve effectively. An MPG_{ii} $(\mathbb{P}, \{V_p\}_{p \in \mathbb{P}}, v_{\text{init}}, \mathbb{D}, E, \{\sim_p\}_{p \in \mathbb{P}}, c)$ is played under hierarchical information if there exists a total order $\prec$ on $\mathbb{P}$ such that for every $p' \prec p$, we have $\sim_p \subseteq \sim_{p'}$. That is, we can order the players such that the information is hierarchical, i.e., player p observes at least as much as all smaller players (w.r.t. $\prec$). By adopting standard techniques, we can show that we can decide if a given group of players can win a game played under hierarchical information [35]–[37].

Lemma 4. *The MPG_{ii} $\mathcal{G}_{\mathcal{K}, \varphi}$ from Definition 2 is played under hierarchical information. Consequently, it is decidable if $\text{wins}(\mathbb{P}_\exists, \mathcal{G}_{\mathcal{K}, \varphi})$.*

V. COMPLETENESS FOR $\exists^* \forall^*$

In our game-based view, we let players construct witness traces for existentially quantified traces. Compared to the HyperLTL semantics, this limits the power of existential quantification. For example, in the semantics of a $\forall \pi_1. \exists \pi_2. \psi$ formula, the choice for π_2 can be based on the *entire* trace assigned to π_1 . In the game-based view, π_2 is constructed step-wise by a player, so the decision in the i th round of the game can only depend on π_1 's prefix of length i . This leads

to incompleteness, i.e., situations where a property holds, but the game is not won by $\mathbb{P}_\exists$.

Example 3 ([17]). Consider the Kripke structure $\mathcal{K}$ in Figure 1 and the HyperLTL formula $\varphi = \forall \pi_1. \exists \pi_2. \Box \Diamond (a_{\pi_2} \leftrightarrow \bigcirc a_{\pi_1})$ which requires that π_2 predicts the next value of π_1 infinitely often. Clearly, $\mathcal{K} \models \varphi$, but $\neg \text{wins}(\{2\}, \mathcal{G}_{\mathcal{K}, \varphi})$. In $\mathcal{G}_{\mathcal{K}, \varphi}$, player 2 has to decide in each step whether AP a should hold but does not know what player 1 will do on trace π_1 in the future. No matter what player 2 does, player 1 can thus always ensure that $a_{\pi_2} \not\leftrightarrow \bigcirc a_{\pi_1}$.

Our game is thus incomplete; already on $\forall^* \exists^*$ properties where our game coincides with the full-information games from [17], cf. Lemma 3. While incomplete for $\forall^* \exists^*$, our game is, perhaps surprisingly, complete for $\exists^* \forall^*$ properties.

Theorem 2. Assume φ is a $\exists^* \forall^*$ HyperLTL formula. Then $\text{wins}(\mathbb{P}_\exists, \mathcal{G}_{\mathcal{K}, \varphi})$ if and only if $\mathcal{K} \models \varphi$.

Note that we can check any $\forall^* \exists^*$ property by checking the negated property (which is $\exists^* \forall^*$). Our game-based approach, therefore, constitutes a sound-and-complete model-checking technique for all HyperLTL formulas with *at most one* quantifier alternation. The cost of this completeness manifests itself in the additional complexity; our game-based approach for $\forall^* \exists^*$ properties (which coincides with [17]) is incomplete but results in a standard game under full information. If we, instead, check the negated $\exists^* \forall^*$ formula, our game is complete, but the game uses partial information, making automated game-solving more challenging.

VI. PROPHECY VARIABLES

For properties beyond $\exists^* \forall^*$, we need additional tools to counteract the incompleteness. In this section, we study the use of prophecy variables in our game-based framework; a technique originally studied in [16], [17] for the verification of $\forall^* \exists^*$ hyperproperties (building on the seminal work by Abadi and Lamport [27]). At a high level, a prophecy provides limited information about the future behavior of other players. For example, in the setting of a $\forall \pi_1. \exists \pi_2. \psi$ formula, the player controlling π_2 only observes the past behavior of the player controlling π_1 . For such a formula, a prophecy is an LTL formula ξ over trace variable π_1 (cf. [17]). In each step of the game, the player controlling π_2 can then query an oracle that tells them whether or not the future behavior of π_1 will satisfy ξ , and base its decision on the additional information provided by the oracle.

Example 4. In Example 3, the player controlling π_2 does not have a winning strategy as it does not know if a will hold on π_1 in the next step. In this example, it suffices for the player controlling π_2 to have access to an oracle that, in each step, predicts whether prophecy $\xi := \bigcirc a_{\pi_1}$ holds (cf. [17]). If ξ holds (so $\bigcirc a_{\pi_1}$), the player can move the π_2 -copy to state s_B (where a holds, cf. Figure 1), thus ensuring that $a_{\pi_2} \leftrightarrow \bigcirc a_{\pi_1}$.

Prophecies essentially combat the problem of having *too little* information about the *future*, which already leads to

incompleteness in the $\forall^* \exists^*$ setting. The main contribution of this paper is that we can use partial information to avoid players having *too much* information about the behavior of (certain) other players, which would lead to *unsoundness*. In this section, we extend the prophecy framework of [17] from $\forall^* \exists^*$ to arbitrary quantifier prefixes. Our prophecy construction combines two ideas: our observation model ensures that players do not observe too much (to ensure soundness), while prophecies provide missing information about future events.

A. Prophecies and Partial Information

To keep notation simple, we assume for the remainder of this section – and w.l.o.g. – that the hyperproperty in question is of the form $\varphi = \forall \pi_1. \exists \pi_2. \forall \pi_3. \dots \forall \pi_{2n-1}. \exists \pi_{2n}. \psi$, i.e., strictly alternates between universal (at odd indices) and existential (at even indices) quantification.

Example 5. Consider the formula

$$\varphi := \forall \pi_1. \exists \pi_2. \forall \pi_3. \exists \pi_4. \Box \Diamond (a_{\pi_2} \leftrightarrow \bigcirc a_{\pi_1}) \wedge \bigcirc \bigcirc (a_{\pi_4} \leftrightarrow \Box (a_{\pi_1} \leftrightarrow a_{\pi_2} \wedge a_{\pi_2} \leftrightarrow a_{\pi_3})).$$

It requires π_2 to globally predict the next step of π_1 , and π_4 should, in the second step, predict whether π_1 , π_2 , and π_3 agree on a . The KS $\mathcal{K}$ in Figure 1 satisfies φ , yet coalition $\{2, 4\}$ loses $\mathcal{G}_{\mathcal{K}, \varphi}$. To win this game, the player controlling π_2 needs (in every step) information about the future of π_1 , and the player controlling π_4 needs (in the second step) information about the joint future of π_1 , π_2 , and π_3 .

B. Prophecies and Prophecy Variables

To ensure soundness, we need to ensure that each player i is (via the prophecies) only given information over traces quantified *before* π_i . For each existentially quantified trace π_{2i} , we therefore track a separate set of prophecies:

Definition 3. A prophecy family is a collection $\vec{\Xi} = \{\Xi_{2i-1}\}_{i=1}^n$, where Ξ_{2i-1} is a finite set of LTL formulas over trace variables from $\{\pi_1, \dots, \pi_{2i-1}\}$.

Intuitively, Ξ_{2i-1} contains all prophecies that provide information to the player controlling trace π_{2i} . Consequently, the formulas in Ξ_{2i-1} only reason about traces $\{\pi_1, \dots, \pi_{2i-1}\}$, which are exactly the traces player i can observe in the game.

Example 6. Consider the property in Example 5. We can construct the prophecy family $\{\Xi_1, \Xi_3\}$, where $\Xi_1 := \{\bigcirc a_{\pi_1}\}$, and $\Xi_3 := \{\Box (a_{\pi_1} \leftrightarrow a_{\pi_2} \wedge a_{\pi_2} \leftrightarrow a_{\pi_3})\}$. These prophecies provide exactly the information needed by players 2 and 4. That is, if players 2 and 4 could, in each step of the game, determine if the prophecies in Ξ_1 and Ξ_3 hold, respectively, they can construct appropriate witness traces and win $\mathcal{G}_{\mathcal{K}, \varphi}$.

Now assume we have a fixed family of LTL formulas $\vec{\Xi} = \{\Xi_{2i-1}\}_{i=1}^n$. The intuition behind the prophecies is that each player π_{2i} is provided with an oracle that – in each step of the game – tells her which of the formulas in Ξ_{2i-1} hold. Following [17], we will use *prophecy variables* to formalize this oracle. A prophecy variable is essentially an AP that we

add to the system, and we ensure that the value of this variable (AP) reflects the truth value of the prophecy formula. For this, we assume that P is a set of prophecy variables for Ξ , i.e., for each $1 \leq i \leq n$ and $\xi \in \Xi_{2i-1}$, there exists a corresponding prophecy variable $p^\xi \in P$. A player can thus query an oracle on whether prophecy ξ currently holds, by simply reading the prophecy variable (AP) p^ξ . The key idea now is that we can attach these prophecy variables to *universally* quantified traces [17]. That is, we let the opposing players $\mathbb{P} \setminus \mathbb{P}_\exists$ (controlling the universally quantified traces) determine the truth value of the prophecy variables, and then ensure, within the HyperLTL formula, that the prophecy variables are set correctly: That is, we modify the HyperLTL formula to ensure that $\xi \in \Xi_{2i-1}$ holds iff the prophecy variable p^ξ is set to true on trace π_{2i-1} . This ensures that the player $2i$ controlling π_{2i} can query the prophecies in Ξ_{2i-1} by looking at the current state of π_{2i-1} (and the Boolean value of the prophecy variables, i.e., AP, in that state), but the players controlling $\pi_1, \dots, \pi_{2i-2}$ cannot.

As a first step, we add the variables in P to the system, which can be set arbitrarily in each step:

Definition 4 ($\mathcal{K}^P$). *Given a KS $\mathcal{K} = (S, s_{init}, \mathbb{D}, \kappa, \ell)$ over AP and a disjoint set of prophecy variables P ($AP \cap P = \emptyset$), define the modified KS $\mathcal{K}^P := (S \times 2^P, s_{init}, \mathbb{D} \times 2^P, \kappa^P, \ell^P)$ over $AP \uplus P$ where for each direction $(d, A) \in \mathbb{D} \times 2^P$, we define κ^P by*

$$\begin{aligned} \kappa^P(s_{init}, (d, A)) &:= (\kappa(s_{init}, d), A) \\ \kappa^P((s, A'), (d, A)) &:= (\kappa(s, d), A) \end{aligned}$$

and $\ell^P(s_{init}) = \ell(s_{init})$ and $\ell^P(s, A) := \ell(s) \cup A$.

Intuitively, $\mathcal{K}^P$ generates all traces of $\mathcal{K}$ extended with all possible evaluations on the prophecy variables.

We then modify the body of the HyperLTL formula such that the original property is only required to hold if all prophecies are set correctly, i.e., a prophecy variable in p^ξ is set to true iff the future traces satisfy ξ .

Definition 5 ($\varphi^{P, \Xi}$). *Given a prophecy family $\Xi = \{\Xi_{2i-1}\}_{i=1}^n$ and a corresponding set of prophecy variables P , define the modified HyperLTL formula $\varphi^{P, \Xi}$ by*

$$\begin{aligned} \varphi^{P, \Xi} &:= \forall \pi_1. \exists \pi_2. \forall \pi_3. \dots \forall \pi_{2n-1}. \exists \pi_{2n}. \\ &\left(\bigcirc \bigcirc \bigwedge_{i=1}^n \left(\bigwedge_{\xi \in \Xi_{2i-1}} ((p^\xi)_{\pi_{2i-1}} \leftrightarrow \xi) \right) \right) \rightarrow \psi. \end{aligned}$$

That is, we require ψ (the original LTL body of φ) to hold, if in every step, for every $1 \leq i \leq n$, and every prophecy formula $\xi \in \Xi_{2i-1}$, ξ holds iff the corresponding prophecy variable p^ξ holds on trace π_{2i-1} . Note how $\varphi^{P, \Xi}$ connects the prophecy variables with the underlying prophecy. If some prophecy variable p^ξ for $\xi \in \Xi_{2i-1}$ is set on (the universally quantified) trace π_{2i-1} , the player $2i \in \mathbb{P}_\exists$ controlling the existentially quantified trace π_{2i} can assume that ξ holds, and vice versa. If this is not the case, i.e., player $2i - 1$ sets p^ξ incorrectly, the premise of $\varphi^{P, \Xi}$ is violated, so the LTL body of $\varphi^{P, \Xi}$ is vacuously true, and $\mathbb{P}_\exists$ wins the game in Definition 2.

Note that trace π_{2i-1} is universally quantified, so we consider all possible valuations of the prophecy variables, including the unique valuation where the prophecy variables are set correctly in each step, i.e., $\bigcirc \bigcirc \bigwedge_{\xi \in \Xi_{2i-1}} ((p^\xi)_{\pi_{2i-1}} \leftrightarrow \xi)$. Observe that we only require the prophecies to be set correctly after the first step (using a single $\bigcirc$), as the unique initial state s_{init} of a KS does not record prophecy variables.

C. Prophecies and Games

We can use the information provided via prophecies in our game-based approach. Instead of checking if the players in $\mathbb{P}_\exists$ win $\mathcal{G}_{\mathcal{K}, \varphi}$, we can then check if they win $\mathcal{G}_{\mathcal{K}^P, \varphi^{P, \Xi}}$. In the latter game, the additional premise in $\varphi^{P, \Xi}$ ensures that the players can use prophecies to peek at future moves of the universal traces.

Example 7. *We consider the example from Examples 5 and 6. We already argued that $\mathbb{P}_\exists = \{2, 4\}$ loses $\mathcal{G}_{\mathcal{K}, \varphi}$. Now consider the prophecy family Ξ from Example 6, with corresponding prophecy variables $P := \{\{p\}, \{pp\}\}$. Using Definition 5, we construct*

$$\begin{aligned} \varphi^{P, \Xi} &= \forall \pi_1. \exists \pi_2. \forall \pi_3. \exists \pi_4. \left[\bigcirc \bigcirc (p_{\pi_1} \leftrightarrow \bigcirc a_{\pi_1}) \wedge \right. \\ &\quad \left. \bigcirc \bigcirc (pp_{\pi_3} \leftrightarrow \bigcirc (a_{\pi_1} \leftrightarrow a_{\pi_2} \wedge a_{\pi_2} \leftrightarrow a_{\pi_3})) \right] \rightarrow \psi, \end{aligned}$$

where ψ is the original LTL body (cf. Example 5). It is easy to see that $\{2, 4\}$ wins $\mathcal{G}_{\mathcal{K}^P, \varphi^{P, \Xi}}$: the prophecy variable p on π_1 hints at the next move of π_1 . If, for example, player 1 sets p to true, player 2 can assume that $\bigcirc a_{\pi_1}$ holds (if it does not, the premise of $\varphi^{P, \Xi}$'s LTL body is violated, and so the play is trivially won by $\{2, 4\}$). Likewise, the prophecy variable pp provides the necessary information for player 4.

D. Soundness

The key result we are left to prove is that the addition of prophecies does not change the HyperLTL semantics, even though the LTL body of $\varphi^{P, \Xi}$ is weaker than the body of φ .

Theorem 3. *Assume a prophecy family $\Xi = \{\Xi_{2i-1}\}_{i=1}^n$ and a corresponding set of prophecy variables P . Then $\mathcal{K} \models \varphi$ if and only if $\mathcal{K}^P \models \varphi^{P, \Xi}$.*

Theorem 3 allows us to soundly combine prophecies with the game-based approach: If $\text{wins}(\mathbb{P}_\exists, \mathcal{G}_{\mathcal{K}^P, \varphi^{P, \Xi}})$, then, by Theorem 1, we have $\mathcal{K}^P \models \varphi^{P, \Xi}$, so, by Theorem 3, we have $\mathcal{K} \models \varphi$. Prophecy variables thus constitute a tool that can strengthen game-based verification in the presence of arbitrary quantifier alternations. This is particularly relevant when using our approach as an interactive proof technique. Once a suitable set of prophecies is found (i.e., a family Ξ s.t. $\mathbb{P}_\exists$ wins $\mathcal{G}_{\mathcal{K}^P, \varphi^{P, \Xi}}$), the prophecies, together with the winning strategies for $\mathbb{P}_\exists$, are an easy-to-check certificate of satisfaction.

VII. RELATED WORK

Logics for Hyperproperties: Most logics for expressing temporal hyperproperties use HyperLTL-style quantification

over execution traces [8]–[10], [38]. In such logics, quantifier alternations are frequently used to, e.g., reason about non-determinism in the system. Our paper proposes a principled approach to deal with quantifier alternations that can be easily extended to other logics that feature HyperLTL-style quantification over system paths/traces.

HyperLTL Verification: Finite-state model-checking of HyperLTL is decidable [3], and complete algorithms rely on expensive automata complementation or language inclusion checks [12], [31]. To approximate this expensive problem, Hsu et al. [39], [40] propose a bounded model-checking approach for HyperLTL by unrolling the system and property into a QBF formula. The other prominent approximation for HyperLTL is the game-based approach [16], [17], which forms the foundation of the present paper. Both approximations are orthogonal to each other. In the game-based approach, we use strategies to resolve existential quantification and can thus reason about temporal behavior along *infinite* paths. In contrast, the QBF-based encoding features the same first-order semantics used in HyperLTL, but bounds the length of the traces, limiting the approach to properties that can be verified or refuted within a bounded timeframe.

Advantages of Game-based Verification: The game-based approach has multiple advantages over automata-complementation-based methods. Firstly, it allows verification in settings where complementation-based approaches fail. For example, the game-based approach can be used to verify *infinite-state* systems by constructing abstract games [41], [42], or utilizing infinite-state game solvers [43]–[47]. Secondly, the game-based approach allows for *interactive proofs*, i.e., the user can manually construct a proof by incrementally defining a winning strategy [18]. This facilitates proofs in situations where automated methods do not scale. And lastly, the game-based approach naturally yields *certificates* of satisfaction, i.e., the computed winning strategy (combined with a set of prophecies, if needed) can easily be checked by independent strategy checkers [19].

In this paper, we propose an extension of the game-based approach to arbitrary quantifier structures, based on the key conceptual contribution of leveraging *partial information*, extending our earlier extended abstract [48]. Based on this key conceptual idea, we provide a sound framework that allows us to utilize the benefits of game-based verification for arbitrary quantifier structures. While solving games under partial information is expensive, our paper also allows for cheaper approaches that build upon the game-based interpretation. For example, we can attempt to find *positional* strategies for $\mathbb{P}_\exists$. Finding positional strategies is much cheaper than finding strategies under perfect recall (i.e., unbounded memory), and, if winning positional strategies for $\mathbb{P}_\exists$ are found, our results allow us to soundly conclude that $\mathcal{K} \models \varphi$. This creates a spectrum of techniques with varying complexity and expressiveness (i.e., in some instances, positional strategies suffice, in others, we might need bounded memory or even perfect recall). In the case of $\exists^*\forall^*$ properties, our definition yields a game where the player acts without any information, a

setting explored extensively within the planning community (cf. *conformant planning*) [33], [49].

Solving Games Under Incomplete Information: We believe that the primary use case of our approach lies in its ability to construct sound-by-design interactive proofs and certificates. Nevertheless, if paired with a solver for games under partial information, our approach could underpin a fully-automated verification pipeline. The study of (multiplayer) games under incomplete information has a long tradition, mostly relying on using a powerset construction or lattice framework to track belief states [24], [25], [35], [36], [50], [51]. Strategy-based logic can explicitly reason about the strategic abilities of players, and extensions to incomplete information exist [20], [23], [52]–[54]. Many of the frameworks used to study incomplete information can be used in our setting: for example, partially observable non-deterministic (POND) planning reasons about an agent that acts in a non-deterministic environment, essentially defining a game under partial information [55], which we can use for hyperproperty verification [33], [56]. Likewise, multi-agent planning [57], [58], partially observable Markov decision processes with multiple agents [59], [60], or multi-agent reinforcement learning [61] study strategy synthesis in partially observable domains.

VIII. CONCLUSION AND FUTURE WORK

In this work, we have proposed a novel hyperproperty verification method using games. In contrast to previous game-based approaches, our method is not limited to $\forall^*\exists^*$ properties but soundly approximates *arbitrary* quantifier structures. Moreover, we designed a prophecy mechanism that aligns with the partial information of the player. Our work creates numerous avenues for future work, both in theory and practice. In theory, it is interesting to study the expressive power of our game-based approximation when using prophecies. Prophecies are a complete proof technique in the setting of $\forall^*\exists^*$ properties, i.e., whenever a property holds, there exists some finite set of ω -regular (not necessarily LTL-definable) prophecies such that the game is won by the verifier [17]. The high-level idea of this construction is to construct prophecies that directly determine successor states, i.e., prophecies $\xi_{s,s'}$ that hold iff, when in state s , moving to s' is the “optimal” move for the verifier (see [17] for details). We conjecture that completeness also holds in the presence of arbitrary quantifier alternations: For a property $Q_1\pi_1 \dots Q_{i-1}\pi_{i-1}\exists\pi_i.Q_{i+1}\pi_{i+1} \dots Q_n\pi_n.\psi$, we can design prophecies that precisely define the optimal behavior for player i (for a fixed system $\mathcal{K}$, $Q_{i+1}\pi_{i+1} \dots Q_n\pi_n.\psi$ is just an ω -regular property over $\pi_1, \dots, \pi_i$). In practice, we can utilize our games as an interactive proof technique, e.g., using the coinductive framework of [18].

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) Grant HYPER (101055412), and by the German Research Foundation (DFG) as part of TRR 248 (389792660).

REFERENCES

- [1] M. R. Clarkson and F. B. Schneider, “Hyperproperties,” in *Computer Security Foundations Symposium, CSF 2008*, 2008.
- [2] A. Pnueli, “The temporal logic of programs,” in *Symposium on Foundations of Computer Science, FOCS 1977*, 1977.
- [3] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, “Temporal logics for hyperproperties,” in *International Conference on Principles of Security and Trust, POST 2014*, 2014.
- [4] S. Zdancewic and A. C. Myers, “Observational determinism for concurrent program security,” in *Computer Security Foundations Workshop, CSFW 2003*, 2003.
- [5] W. van der Hoek and M. J. Wooldridge, “Model checking knowledge and time,” in *International Workshop on Model Checking of Software, SPIN 2002*, 2002.
- [6] R. Beutner, B. Finkbeiner, H. Frenkel, and N. Metzger, “Second-order hyperproperties,” in *International Conference on Computer Aided Verification, CAV 2023*, 2023.
- [7] J. Y. Halpern, “Reasoning about knowledge: An overview,” in *Theoretical aspects of reasoning about knowledge*, 1986.
- [8] M. N. Rabe, “A temporal logic approach to information-flow control,” Ph.D. dissertation, Saarland University, 2016.
- [9] J. O. Gutsfeld, M. Müller-Olm, and C. Ohrem, “Propositional dynamic logic for hyperproperties,” in *International Conference on Concurrency Theory, CONCUR 2020*, 2020.
- [10] N. Coenen, B. Finkbeiner, C. Hahn, and J. Hofmann, “The hierarchy of hyperlogics,” in *Symposium on Logic in Computer Science, LICS 2019*, 2019.
- [11] G. Barthe, P. R. D’argenio, and T. Rezk, “Secure information flow by self-composition,” *Math. Struct. Comput. Sci.*, 2011.
- [12] B. Finkbeiner, M. N. Rabe, and C. Sánchez, “Algorithms for model checking HyperLTL and HyperCTL*,” in *International Conference on Computer Aided Verification, CAV 2015*, 2015.
- [13] Y. Wang, S. Nalluri, and M. Pajic, “Hyperproperties for robotics: Planning via hyperltl,” in *International Conference on Robotics and Automation, ICRA 2020*, 2020.
- [14] S. Biewer, R. Dimitrova, M. Fries, M. Gazda, T. Heinze, H. Hermanns, and M. R. Mousavi, “Conformance relations and hyperproperties for doping detection in time and space,” *Log. Methods Comput. Sci.*, 2022.
- [15] M. Anand, V. Murali, A. Trivedi, and M. Zamani, “Verification of hyperproperties for dynamical systems via barrier certificates,” *IEEE Transactions on Automatic Control*, 2024.
- [16] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup, “Verifying hyperliveness,” in *International Conference on Computer Aided Verification, CAV 2019*, 2019.
- [17] R. Beutner and B. Finkbeiner, “Prophecy variables for hyperproperty verification,” in *Computer Security Foundations Symposium, CSF 2022*, 2022.
- [18] A. Correnson and B. Finkbeiner, “Coinductive proofs for temporal hyperliveness,” *Proc. ACM Program. Lang.*, no. POPL, 2025.
- [19] R. Beutner, B. Finkbeiner, and A. Göbl, “Visualizing game-based certificates for hyperproperty verification,” in *International Symposium on Formal Methods, FM 2024*, 2024.
- [20] R. Berthon, B. Maubert, and A. Murano, “Decidability results for ATL* with imperfect information and perfect recall,” in *Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017*, 2017.
- [21] D. Berwanger, A. B. Mathew, and M. van den Bogaard, “Hierarchical information and the synthesis of distributed strategies,” *Acta Informatica*, 2018.
- [22] B. Maubert and A. Murano, “Reasoning about knowledge and strategies under hierarchical information,” in *International Conference on Principles of Knowledge Representation and Reasoning, KR 2018*, 2018.
- [23] R. Berthon, B. Maubert, A. Murano, S. Rubin, and M. Y. Vardi, “Strategy logic with imperfect information,” *ACM Trans. Comput. Log.*, 2021.
- [24] D. Berwanger, K. Chatterjee, M. D. Wulf, L. Doyen, and T. A. Henzinger, “Strategy construction for parity games with imperfect information,” *Inf. Comput.*, 2010.
- [25] K. Chatterjee and L. Doyen, “The complexity of partial-observation parity games,” in *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2010*, 2010.
- [26] P. Gastin, N. Sznajder, and M. Zeitoun, “Distributed synthesis for well-connected architectures,” *Formal Methods Syst. Des.*, 2009.
- [27] M. Abadi and L. Lamport, “The existence of refinement mappings,” in *Symposium on Logic in Computer Science, LICS 1988*, 1988.
- [28] R. Beutner and B. Finkbeiner, “On hyperproperty verification, quantifier alternations, and games under partial information,” *CoRR*, 2025.
- [29] J. Esparza, J. Kretínský, J. Raskin, and S. Sickert, “From LTL and limit-deterministic büchi automata to deterministic parity automata,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017*, 2017.
- [30] N. Piterman, “From nondeterministic büchi and streett automata to deterministic parity automata,” *Log. Methods Comput. Sci.*, 2007.
- [31] R. Beutner and B. Finkbeiner, “AutoHyper: Explicit-state model checking for HyperLTL,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023*, 2023.
- [32] V. Malvone, A. Murano, and L. Sorrentino, “Concurrent multi-player parity games,” in *International Conference on Autonomous Agents & Multiagent Systems, AAMAS 2016*, 2016.
- [33] R. Beutner and B. Finkbeiner, “On conformant planning and model-checking of $\exists^*\forall^*$ hyperproperties,” in *European Conference on Artificial Intelligence, ECAI 2025*, 2025.
- [34] A. Pnueli and R. Rosner, “Distributed reactive systems are hard to synthesize,” in *Symposium on Foundations of Computer Science, FOCS 1990*, 1990.
- [35] J. H. Reif, “The complexity of two-player games of incomplete information,” *J. Comput. Syst. Sci.*, 1984.
- [36] G. Peterson, J. Reif, and S. Azhar, “Decision algorithms for multiplayer noncooperative games of incomplete information,” *Computers & Mathematics with Applications*, 2002.
- [37] B. Finkbeiner and S. Schewe, “Uniform distributed synthesis,” in *Symposium on Logic in Computer Science, LICS 2005*, 2005.
- [38] L. V. Nguyen, J. Kapinski, X. Jin, J. V. Deshmukh, and T. T. Johnson, “Hyperproperties of real-valued signals,” in *International Conference on Formal Methods and Models for System Design, MEMOCODE 2017*, 2017.
- [39] T. Hsu, C. Sánchez, and B. Bonakdarpour, “Bounded model checking for hyperproperties,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021*, 2021.
- [40] T. Hsu, C. Sánchez, S. Sheinvald, and B. Bonakdarpour, “Efficient loop conditions for bounded model checking hyperproperties,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023*, 2023.
- [41] R. Beutner and B. Finkbeiner, “Software verification of hyperproperties beyond k-safety,” in *International Conference on Computer Aided Verification, CAV 2022*, 2022.
- [42] S. Itzhaky, S. Shoham, and Y. Vazel, “Hyperproperty verification as CHC satisfiability,” in *European Symposium on Programming, ESOP 2024*, 2024.
- [43] P. Battigalli, “Rationalizability in infinite, dynamic games with incomplete information,” *Research in Economics*, 2003.
- [44] L. de Alfaro, T. A. Henzinger, and R. Majumdar, “Symbolic algorithms for infinite-state games,” in *International Conference on Concurrency Theory, CONCUR 2001*, 2001.
- [45] P. Heim and R. Dimitrova, “Solving infinite-state games via acceleration,” *Proc. ACM Program. Lang.*, no. POPL, 2024.
- [46] M. Faella and G. Parlato, “Reachability games modulo theories with a bounded safety player,” in *Conference on Artificial Intelligence, AAAI 2023*, 2023.
- [47] C. Baier, N. Coenen, B. Finkbeiner, F. Funke, S. Jantsch, and J. Siber, “Causality-based game solving,” in *International Conference on Computer Aided Verification, CAV 2021*, 2021.
- [48] R. Beutner and B. Finkbeiner, “Multiplayer games with incomplete information for hyperproperty verification,” in *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025*, 2025.
- [49] R. P. Goldman and M. S. Boddy, “Expressive planning and explicit knowledge,” in *International Conference on Artificial Intelligence Planning Systems, AIPS 1996*, 1996.
- [50] J. Raskin, K. Chatterjee, L. Doyen, and T. A. Henzinger, “Algorithms for omega-regular games with imperfect information,” *Log. Methods Comput. Sci.*, 2007.
- [51] M. D. Wulf, L. Doyen, and J. Raskin, “A lattice theory for solving games of imperfect information,” in *International Workshop on Hybrid Systems: Computation and Control, HSCC 2006*, 2006.
- [52] R. Beutner and B. Finkbeiner, “Hyper strategy logic,” in *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024*, 2024.
- [53] J. Pilecki, M. A. Bednarczyk, and W. Jamroga, “Model checking properties of multi-agent systems with imperfect information and imperfect recall,” in *International Conference on Intelligent Systems*, 2014.

- [54] R. Beutner and B. Finkbeiner, “Strategy logic, imperfect information, and hyperproperties,” in *International Conference on Principles of Knowledge Representation and Reasoning, KR 2025*, 2025.
- [55] R. I. Brafman, G. Shani, and S. Zilberstein, “Qualitative planning under partial observability in multi-agent domains,” in *Conference on Artificial Intelligence, AAAI 2013*, 2013.
- [56] R. Beutner and B. Finkbeiner, “Non-deterministic planning for hyperproperty verification,” in *International Conference on Automated Planning and Scheduling, ICAPS 2024*, 2024.
- [57] P. J. Gmytrasiewicz and P. Doshi, “A framework for sequential planning in multi-agent settings,” *J. Artif. Intell. Res.*, 2005.
- [58] M. F. L. Galesloot, T. D. Simão, S. Junges, and N. Jansen, “Factored online planning in many-agent POMDPs,” in *Conference on Artificial Intelligence, AAAI 2024*, 2024.
- [59] C. Zhang and V. R. Lesser, “Coordinated multi-agent reinforcement learning in networked distributed POMDPs,” in *Conference on Artificial Intelligence, AAAI 2011*, 2011.
- [60] C. Amato and F. A. Oliehoek, “Scalable planning and learning for multiagent POMDPs,” in *Conference on Artificial Intelligence, AAAI 2015*, 2015.
- [61] L. Busoniu, R. Babuska, and B. D. Schutter, “A comprehensive survey of multiagent reinforcement learning,” *IEEE Trans. Syst. Man Cybern. Part C*, 2008.

Scalable MLTL Runtime Monitoring and Satisfiability via Bit-Vector Encoding

Christopher Johannsen 
Iowa State University
Ames, United States
cgjohann@iastate.edu

Phillip H. Jones 
Iowa State University
Ames, United States
phjones@iastate.edu

Kristin Yvonne Rozier 
Iowa State University
Ames, United States
kyrozier@iastate.edu

Tichakorn Wongpiromsarn 
Iowa State University
Ames, United States
nok@iastate.edu



Abstract—We introduce a novel encoding for Mission-time Linear Temporal Logic (MLTL), inspired by the recently-proposed Temporal Exponential Propagation method for learning finite-trace Linear Temporal Logic (LTL_f) formulas. Our encoding translates MLTL formulas into bit-vector expressions whose size is $\mathcal{O}(n \log^2(K))$ with n and K being proportional to the formula size and largest temporal interval present in the formula respectively. We demonstrate how this encoding can be leveraged for two core applications: satisfiability checking and runtime monitoring. For satisfiability, our approach complements existing techniques by efficiently solving instances that are challenging for traditional encodings. For runtime monitoring, we show that our method achieves significantly higher throughput than current state-of-the-art tools, enabling faster and more scalable runtime verification in time-sensitive systems. Experimental results confirm that the proposed encoding and monitoring algorithm provide a practical and effective foundation for reasoning about MLTL properties, achieving 10.52 times higher throughput compared to state-of-the-art online techniques over a sample of pattern-instantiated formulas.

I. INTRODUCTION

Modern high-throughput systems, such as those found in autonomous vehicles, networking, and large-scale industrial automation, must process and react to vast data streams with minimal latency. In these settings, correctness is often time-sensitive, and verification must occur at the system’s speed. Traditional techniques for specifying and monitoring temporal properties frequently become performance bottlenecks, unable to scale with the volume and velocity of data [1], [2]. As systems grow more complex and their responsiveness more critical, the need for lightweight, efficient reasoning frameworks that can keep up with high-speed streams becomes increasingly urgent.

Mission-time Linear Temporal Logic (MLTL) [3] is well-suited for specifying and monitoring the behavior of real-time systems, offering the ability to express constraints over bounded time intervals. Its expressiveness has led to its adoption in several applications, including cyber-physical systems and safety-critical embedded software [4], [5], [6], [7], [8]. Two fundamental reasoning tasks in MLTL are satisfiability

checking, determining whether a formula can be satisfied by any trace; and runtime monitoring, checking whether a given trace satisfies a formula. While there has been progress on MLTL satisfiability, existing methods still struggle to handle all instances effectively [9], highlighting the need for more scalable and efficient techniques for MLTL reasoning.

Stream-based monitoring of temporal logics such as MLTL, Metric Temporal Logic (MTL) [10], [11], Metric Dynamic Logic (MDL) [12], and Metric First-Order Temporal Logic [13] have led to the development of various tools. MONPOLY [14] and DEJAVU [15] target high-throughput settings, leveraging first-order temporal logics to monitor system behavior through database-like event streams. However, their reliance on a first-order framework makes them less suitable for lightweight, non-first-order specifications. In contrast, tools such as AERIAL [16] and HYDRA [17] focus on monitoring MTL and MDL formulas, but HYDRA operates exclusively offline. While numerous stream-based monitoring frameworks exist, including LOLA/RTLOLA [18], [19], COPLOT [20], and TESSLA [21], these support a range of tool-specific, stream-based specification languages that favor expressiveness over performance. In contrast, R2U2 optimizes for real-time performance in tightly-resource-constrained environments [3], [22], [23], [24]. We narrow our focus on the sweet spot in-between: stream-based monitors that target temporal logic specifications, aiming to improve performance in high-throughput, real-time scenarios.

We present a new approach to encoding MLTL formulas inspired by Temporal Exponential Propagation, a recently-proposed technique for learning finite-trace Linear Temporal Logic (LTL_f) formulas [25]. Our approach translates MLTL formulas into bit-vector expressions whose size is logarithmic in the size of the original formula. Our encoding uses the SMT theory of quantifier-free bit-vectors for satisfiability checking, offering a complementary alternative to existing quantified, linear integer arithmetic methods. This encoding enables a portfolio of solvers to solve 62% of pseudo-randomly generated instances faster than previous approaches. For runtime

monitoring, the use of branch-free bit-vector operations results in significant throughput gains, making it well-suited for systems with high-frequency data streams. Experimental results show that our approach consistently outperforms the throughput of state-of-the-art MLTL online monitoring techniques, on average by 10.52 times over a set of five pattern-instantiated formulas from literature [26].

Contributions We contribute a novel encoding for MLTL formulas into bit-vector expressions, prove its correctness, and show its complexity is $\mathcal{O}(n \log^2(K))$ in the total number of bit-vector operations with n and K being proportional to the formula size and largest temporal interval present in the formula respectively. We also provide an open-source implementation of both an MLTL satisfiability checker and runtime monitor, named SABRE (Stream Analysis via Bit-wise Reasoning), and an experimental evaluation empirically demonstrating the performance of each [27]¹.

Organization Section II introduces preliminaries, including important previously-proved properties of the logic MLTL and the bit-vector operations upon which we build. In Section III, we present the new bit-vector encoding for MLTL formulas along with correctness and complexity proofs. Section IV applies this encoding to the MLTL satisfiability problem and includes corresponding experimental results. In Section V, we apply the approach to stream-based MLTL runtime monitoring, describing the monitoring algorithm and evaluating its performance. Finally, Section VI concludes the paper and outlines directions for future work.

II. PRELIMINARIES

First, let $[l, u] = \{n \in \mathbb{N}_0 \mid l \leq n \leq u\}$ be an inclusive interval where l, u are zero-inclusive naturals ($\mathbb{N}_0$). The size of an interval $[l, u]$ is defined as $|[l, u]| = u - l + 1$. Let $\mathbf{T}$ and $\mathbf{F}$ denote logical truth and false respectively.

A. Mission-time Linear Temporal Logic (MLTL)

We proceed with defining MLTL [3].

Definition 1 (MLTL Syntax). The syntax of an MLTL formula over a set of atomic propositions $\mathcal{AP}$ where $p \in \mathcal{AP}$ and φ, ψ are MLTL formulas is defined as:

$$p \mid \neg\varphi \mid \varphi \wedge \psi \mid \Diamond_{[l,u]}\varphi \mid \varphi \mathcal{U}_{[l,u]}\psi$$

A useful metric for bounding the size of transformed formula is the *closure* [9] of an MLTL formula φ , denoted $cl(\varphi)$, which captures the set of syntactic elements of φ (i.e., elements in the DAG of φ), ignoring temporal intervals. We formally define this set as follows:

$$\begin{aligned} cl(p) &= \{p\}, & cl(\neg\varphi) &= \{\neg\varphi\} \cup cl(\varphi) \\ cl(\varphi \wedge \psi) &= \{\varphi \wedge \psi\} \cup cl(\varphi) \cup cl(\psi) \\ cl(\Diamond_{[l,u]}\varphi) &= \{\Diamond_{[l,u]}\varphi\} \cup cl(\varphi) \\ cl(\varphi \mathcal{U}_{[l,u]}\psi) &= \{\varphi \mathcal{U}_{[l,u]}\psi\} \cup cl(\varphi) \cup cl(\psi) \end{aligned}$$

We define a related notion of temporal closure, denoted $cl_T(\varphi)$, to capture only the temporal operators in the DAG of φ :

$$cl_T(\varphi) = \{\psi \in cl(\varphi) \mid \psi = \Diamond_{[l,u]}\xi \text{ or } \psi = \xi \mathcal{U}_{[l,u]}\vartheta\}$$

MLTL formulas are interpreted over finite traces. A trace π is a finite sequence of states with length $|\pi| \in \mathbb{N}_0$, where each state defines a set of atomic propositions that are true in that state. We denote the state at timestamp $0 \leq \tau < |\pi|$ by $\pi[\tau] \subseteq \mathcal{AP}$ and π_τ the trace π starting at and including the state at τ . The concatenation of two traces π_1 and π_2 is denoted $\pi_1 \cdot \pi_2$.

Definition 2 (MLTL Semantics). The satisfaction of an MLTL formula by a trace π at timestamp τ is defined recursively as:

$$\begin{aligned} \pi, \tau &\models p \text{ iff } p \in \pi[\tau] & \pi, \tau &\models \neg\varphi \text{ iff } \pi, \tau \not\models \varphi \\ \pi, \tau &\models \varphi \wedge \psi \text{ iff } \pi, \tau \models \varphi \text{ and } \pi, \tau \models \psi \\ \pi, \tau &\models \Diamond_{[l,u]}\varphi \text{ iff } \tau + l < |\pi| \text{ and } \pi, i \models \varphi \text{ for some } i < |\pi| \\ &\text{such that } l + \tau \leq i \leq u + \tau \\ \pi, \tau &\models \varphi \mathcal{U}_{[l,u]}\psi \text{ iff } \tau + l < |\pi| \text{ and } \pi, i \models \psi \text{ for some } i < |\pi| \\ &\text{such that } l + \tau \leq i \leq u + \tau \text{ and } \pi, j \models \varphi \text{ for all } j \\ &\text{such that } l + \tau \leq j < i \end{aligned}$$

We say two MLTL formulas φ, ψ are *semantically equivalent* iff $\pi, \tau \models \varphi \leftrightarrow \pi, \tau \models \psi$ for all finite traces π over $\mathcal{AP}$ and timestamps $0 \leq \tau < |\pi|$. MLTL keeps the standard operator equivalences from LTL, including $\Diamond_{[l,u]}\varphi = (\mathbf{T} \mathcal{U}_{[l,u]}\varphi)$, $\varphi \mathcal{R}_{[l,u]}\psi = \neg(\neg\varphi \mathcal{U}_{[l,u]}\neg\psi)$, and $\Box_{[l,u]}\varphi = (\mathbf{F} \mathcal{R}_{[l,u]}\varphi)$. To complete the MLTL semantics, we define $\mathbf{F} = \neg\mathbf{T}$, $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, and $\neg\Diamond_{[l,u]}\varphi = \Box_{[l,u]}\neg\varphi$.

We also define the *computation length* [28] of MLTL formulas.

Definition 3 (Computation Length). Let $p \in \mathcal{AP}$ be an atomic proposition and φ, ψ be MLTL formulas. Then the *computation length* of an MLTL formula is defined recursively as follows:

$$\begin{aligned} cplen(p) &= 1 & cplen(\neg\varphi) &= cplen(\varphi) \\ cplen(\varphi \wedge \psi) &= \max(cplen(\varphi), cplen(\psi)) \\ cplen(\Diamond_{[l,u]}\psi) &= u + cplen(\psi) \\ cplen(\varphi \mathcal{U}_{[l,u]}\psi) &= u + \max(cplen(\varphi) - 1, cplen(\psi)) \end{aligned}$$

This definition captures the notion of the minimum length of trace such that no extension to that length trace could change the satisfaction of the formula. This is similar to the definition of *safety properties*, which states that no extension to any counterexample to the safety property will cause the extended counterexample to satisfy the property — once the property is violated it cannot be un-violated. Similarly, once a trace is of length $cplen(\varphi)$, no extension can change its satisfaction of φ . This is formalized in the following (Corollary `trace_append_iff` of [29]):

Corollary 1 (Computation Length Correctness). For an MLTL formula φ and trace π with $|\pi| \geq cplen(\varphi)$, appending any suffix π' to π will not affect whether φ holds over π starting at index 0, i.e.,

$$\pi, 0 \models \varphi \text{ iff } \forall \pi'. ((\pi \cdot \pi'), 0 \models \varphi).$$

¹Artifact is available at <https://doi.org/10.5281/zenodo.16884213>

Two problems of interest for MLTL are the *satisfiability* and *monitoring* problems. Satisfiability checking of an MLTL formula φ asks whether there exists a trace π such that $\pi, 0 \models \varphi$. Monitoring an MLTL formula φ over trace π asks to compute $\pi, \tau \models \varphi$ for all τ . We consider *stream-based* monitoring, where the input is an unbounded sequence of states (i.e., a stream) so that the trace considered at time τ is $\pi[0], \pi[1], \dots, \pi[\tau]$ and the output is a sequence of verdicts corresponding to $(\pi, 0 \models \varphi), (\pi, 1 \models \varphi), \dots, (\pi, \tau' \models \varphi)$ where $\tau' \leq \tau$. In this context, we ignore the trace length constraints of the MLTL semantics, i.e., if $\pi, \tau \not\models \varphi$ $\mathcal{U}_{[l,u]} \psi$ trivially holds because $\tau + l \geq |\pi|$, then we choose instead to not produce a verdict for τ since we always expect more states to populate π from the input stream.

B. Bit-vector Operations

We consider bit-vectors as a finite sequence of 1s and 0s with a fixed length. A bit-vector b has a length, denoted $|b|$. We consider standard bit-vector operations, including negation ($\sim$), and ($\&$), or ($|$), left-shift ($\ll$), and or-reduction (redor). We write constant bit-vectors in the form $0b[01]^+$ with the left-most bit being the least-significant bit (LSB). For example, $0b0101$ is a bit-vector of length 4 and decimal value 10. We allow indexing of bit-vectors, so that $b[i]$ denotes the i th bit from the left of b (e.g., $b[0]$ is the LSB of b) and $b[i : j]$ denotes the extraction of bits i to j , inclusive, from b with $i \leq j$, producing a bit-vector of width $j - i + 1$. With slight abuse of notation, we also allow propositional operators on bit-vectors of length 1, for example, $b[i] \wedge b[j]$, and also let $T \equiv 1 \equiv 0b1$ and $F \equiv 0 \equiv 0b0$. We also define the term $\text{zeroes}(w)$ with width $w \in \mathbb{N}_0$ to denote the bit-vector of width w that is 0 at every index, e.g., $\text{zeroes}(3) = 0b000$. The result of shifting a bit-vector by a value greater than or equal to the width of the bit-vector is the zero bit-vector of that width: $b \ll x = \text{zeroes}(|b|)$ if $x \geq |b|$.

III. MLTL BIT-VECTOR ENCODING

The intuition for the MLTL bit-vector encoding is to consider a trace as a set of bit-vectors, one for each proposition. Then, each MLTL operator can be encoded as a bit-vector operation e.g., $\neg$ as $\sim$, $\vee$ as $|$, and $\wedge$ as $\&$. We can encode each temporal operator as a sequence of shifts logarithmic in size, but only if the operator's interval's size is a power of 2. We introduce a fragment of MLTL that considers only these operators, then show that any MLTL formula has an equivalent formula in this fragment.

Definition 4. MLTL₂ is the fragment of MLTL formulas where all temporal operators have intervals whose sizes are powers of two, i.e., if $p \in \mathcal{AP}$ and φ and ψ are MLTL₂ formulas then so are

$$p \mid \neg\varphi \mid \varphi \wedge \psi \mid \Diamond_{[l,u]}\varphi \mid \varphi \mathcal{U}_{[l,u]}\psi$$

where $u - l + 1 = 2^k$ for some $k \in \mathbb{N}_0$. In terms of k , the $\Diamond$ and $\mathcal{U}$ operators must be of the form $\Diamond_{[l, l+2^k-1]}$ and $\mathcal{U}_{[l, l+2^k-1]}$.

Example MLTL₂ formulas include $\Diamond_{[0,7]}p$, $p \mathcal{U}_{[2,17]} q$, and $\square_{[5,5]}p$. We now define a function $d()$ for converting MLTL

formulas into equivalent MLTL₂ formulas by *decomposing* each temporal formula into a nested sequence of temporal operators, where each operator's interval length is expressed as a sum of powers of two. Given a temporal operator with interval $[l, u]$, let $k = \max\{k \in \mathbb{N}_0 \mid l + (2^k - 1) \leq u\}$ be the maximum (one less) power of 2 that is less than or equal to u . Then we define d as follows:

$$\begin{aligned} d(p) &= p & d(\neg\varphi) &= \neg d(\varphi) & d(\varphi \wedge \psi) &= d(\varphi) \wedge d(\psi) \\ d(\Diamond_{[l,u]}\psi) &= \Diamond_{[l, l+(2^k-1)]} d(\Diamond_{[0, (u-l)-(2^k-1)]}\psi) \\ d(\psi \mathcal{U}_{[l,u]}\xi) &= \psi \mathcal{U}_{[l, l+(2^k-1)]} (d(\psi \mathcal{U}_{[0, (u-l)-(2^k-1)]}\xi)) \end{aligned}$$

To complete the definition of d and ensure that d terminates, let $d(\Diamond_{[0,0]}\psi) = \psi$ and $d(\psi \mathcal{U}_{[0,0]}\xi) = \xi$. Intuitively, $d()$ creates a nesting of temporal operators whose sum of lower bounds and sum of upper bounds equals to the original operator's bounds. We also note that since d always uses the largest possible k to decompose each interval, the nested sequence of temporal operators is minimal in size. In fact, the number of introduced temporal operators is equal to the number of 1s in the original interval size's binary representation.

Example 1. Applying d to the following examples, we obtain equivalent MLTL₂ formulas:

$$\begin{aligned} d(\Diamond_{[0,6]}p) &= \Diamond_{[0,3]} d(\Diamond_{[0,3]}p) \\ &= \Diamond_{[0,3]} (\Diamond_{[0,3]} d(\Diamond_{[0,0]}p)) \\ &= \Diamond_{[0,3]} (\Diamond_{[0,3]} p) \\ d(\Diamond_{[2,15]}p) &= \Diamond_{[2,9]} d(\Diamond_{[0,6]}p) = \Diamond_{[2,9]} (\Diamond_{[0,3]} (\Diamond_{[0,3]} p)) \\ d(p \mathcal{U}_{[2,15]} q) &= p \mathcal{U}_{[2,9]} d(p \mathcal{U}_{[0,6]} q) \\ &= p \mathcal{U}_{[2,9]} (p \mathcal{U}_{[0,3]} (d(p \mathcal{U}_{[0,3]} q))) \\ &= p \mathcal{U}_{[2,9]} (p \mathcal{U}_{[0,3]} (p \mathcal{U}_{[0,3]} d(p \mathcal{U}_{[0,0]} q))) \\ &= p \mathcal{U}_{[2,9]} (p \mathcal{U}_{[0,3]} (p \mathcal{U}_{[0,3]} q)) \end{aligned}$$

Lemma 1 (Future Equivalence). *For any MLTL formula φ , $\Diamond_{[l_1, u_1]} \Diamond_{[l_2, u_2]} \varphi \equiv \Diamond_{[l_1+l_2, u_1+u_2]} \varphi$.*

Proof. Theorem 1 of [30]. □

Lemma 2 (Until Equivalence). *For any MLTL formulas φ and ψ , $\varphi \mathcal{U}_{[l, u_1]} (\varphi \mathcal{U}_{[0, u_2]} \psi) \equiv \varphi \mathcal{U}_{[l, u_1+u_2]} \psi$.*

Proof. Theorem 3 of [28]. □

Before we proceed with proving the correctness and complexity of $d()$, we define the sequence $S(\varphi)$ for MLTL formula φ as the sequence of values $2^k - 1$ for each of the d -introduced operators given an MLTL formula φ (using $\cdot$ for concatenation and $k = \max\{k \in \mathbb{N}_0 \mid l + (2^k - 1) \leq u\}$):

$$S(p) = S(\neg\varphi) = S(\varphi \wedge \psi) = []$$

$$S(\Diamond_{[0,0]}\psi) = S(\psi \mathcal{U}_{[0,0]}\xi) = []$$

$$S(\Diamond_{[l,u]}\psi) = [2^k - 1] \cdot S(\Diamond_{[0, (u-l)-(2^k-1)]}\psi)$$

$$S(\psi \mathcal{U}_{[l,u]}\xi) = [2^k - 1] \cdot S(\psi \mathcal{U}_{[0, (u-l)-(2^k-1)]}\xi)$$

Using the formulas from Example 1, $S(\Diamond_{[0,6]}p) = [3, 3]$ and $S(\Diamond_{[2,15]}p) = S(p \mathcal{U}_{[2,15]} q) = [7, 3, 3]$.

Lemma 3 (Correctness and Complexity of $d()$). *An MLTL formula φ with maximal temporal interval size K has an equivalent MLTL₂ formula ψ with $|cl(\psi)| \leq |cl(\varphi)| \cdot \log_2(K)$.*

Proof. We show that $d()$ produces such an MLTL₂ formula. The output of $d(\varphi)$ is an MLTL₂ formula by inspection since for each introduced temporal operator, its interval $[l, l+2^k-1]$ has size $(l+2^k-1) - l + 1 = 2^k$.

We now show $\varphi \equiv d(\varphi)$ for any MLTL formula φ via structural induction over φ . In the base case, $\varphi = p$ where $d(p) = p$ so that $d(p) \equiv p$. If $\varphi = \neg\psi$ or $\varphi = \psi \wedge \xi$, then $d(\neg\psi) \equiv \neg d(\psi)$ and $d(\psi \wedge \xi) \equiv d(\psi) \wedge d(\xi)$ by the inductive hypotheses $d(\psi) \equiv \psi$ and $d(\xi) \equiv \xi$.

Now, let $\varphi = \Diamond_{[l,u]}\psi$ and inductively assume $d(\psi) \equiv \psi$. Let s_i be the i th value in $S(\varphi)$. Applying $d()$ to φ and using Lemma 1:

$$\begin{aligned} d(\Diamond_{[l,u]}\psi) &= \Diamond_{[l,l+s_1]}d(\Diamond_{[0,(u-l)-s_1]}\psi) \\ &= \Diamond_{[l,l+s_1]}\Diamond_{[0,s_2]}\dots\Diamond_{[0,s_n]}d(\psi) \\ &\equiv \Diamond_{[l,l+s_1+s_2]}\dots\Diamond_{[0,s_n]}d(\psi) \\ &\equiv \Diamond_{[l,u]}d(\psi) \equiv \Diamond_{[l,u]}\psi \end{aligned}$$

where $u = l + s_1 + s_2 + \dots + s_n$ by the definition of $d()$.

We apply a similar argument for $\varphi = \psi \mathcal{U}_{[l,u]}\xi$, where we inductively assume $d(\psi) \equiv \psi$ and $d(\xi) \equiv \xi$. Applying $d()$ to φ and using Lemma 2:

$$\begin{aligned} d(\psi \mathcal{U}_{[l,u]}\xi) &= d(\psi) \mathcal{U}_{[l,l+s_1]}d(\psi \mathcal{U}_{[0,(u-l)-s_1]}\xi) \\ &= d(\psi) \mathcal{U}_{[l,l+s_1]}(d(\psi) \mathcal{U}_{[0,s_2]}(\dots d(\psi) \mathcal{U}_{[0,s_n]}d(\xi))) \\ &\equiv d(\psi) \mathcal{U}_{[l,l+s_1+s_2]}(\dots d(\psi) \mathcal{U}_{[0,s_n]}d(\xi)) \\ &\equiv d(\psi) \mathcal{U}_{[l,u]}d(\xi) \equiv \psi \mathcal{U}_{[l,u]}\xi \end{aligned}$$

where $u = l + s_1 + s_2 + \dots + s_n$ by the definition of $d()$.

We now bound the size of the closure of the formula produced by $d()$ in terms of the largest interval size in φ . Let ψ be the sub-formula of φ with this largest temporal interval size, i.e., the formula associated with $K = \max\{|[l, u]| \mid \psi \in cl_T(\varphi)\}$, and let $[l, u]$ be this interval. Applying $d()$ to ψ introduces at most $|S(\psi)| \leq \log_2(K)$ new operators, and since K is maximal, we have $|cl(d(\varphi))| \leq |cl(\varphi)| \cdot \log_2(K)$. $\square$

Note that although $d(\psi \mathcal{U}_{[l,u]}\xi)$ may introduce copies of ψ , the fact that the closure represents the sub-formulas in the DAG representation of the formula ensures that these copies do not contribute to the complexity of the encoding.

We now define an MLTL bit-vector encoding (MLTL-BV), inspired by the Temporal Exponential Propagation method proposed in [25].

Definition 5 (MLTL-BV Encoding). Let φ, ψ be MLTL₂² formulas and π be a trace, then we define a bit-vector $bv(\varphi, \pi)$

²The decomposition via d may not be necessary; in case the interval is a non-power-of-two the final shift may be by the “leftover” amount. Proofs and implementation of this approach is left to future work.

of width $|\pi|$ as follows where $k \in \mathbb{N}_0$ and using $ite(b, x, y)$ as the if-then-else operator:

$$\begin{aligned} bv(p, \pi)[\tau] &= ite(p \in \pi[\tau], 1, 0) \text{ for } 0 \leq \tau < |\pi| \\ bv(\neg\varphi, \pi) &= \sim bv(\varphi, \pi) \\ bv(\varphi \wedge \psi, \pi) &= bv(\varphi, \pi) \& bv(\psi, \pi) \\ bv(\Diamond_{[l,l+2^k-1]}\varphi, \pi) &= \text{let } b_0 = bv(\varphi, \pi) \lll l, \\ &\quad b_1 = b_0 \mid (b_0 \lll 1), \\ &\quad b_2 = b_1 \mid (b_1 \lll 2), \\ &\quad b_3 = b_2 \mid (b_2 \lll 4), \\ &\quad \dots \\ &\quad b_k = b_{k-1} \mid (b_{k-1} \lll 2^{k-1}) \text{ in } b_k \\ bv(\varphi \mathcal{U}_{[l,l+2^k-1]}\psi, \pi) &= \text{let} \\ &\quad b_{\psi_0} = bv(\psi, \pi) \lll l, \quad b_{\varphi_0} = bv(\varphi, \pi) \lll l \\ &\quad b_{\psi_1} = b_{\psi_0} \mid (b_{\varphi_0} \& (b_{\psi_0} \lll 1)), \quad b_{\varphi_1} = b_{\varphi_0} \& (b_{\varphi_0} \lll 1), \\ &\quad b_{\psi_2} = b_{\psi_1} \mid (b_{\varphi_1} \& (b_{\psi_1} \lll 2)), \quad b_{\varphi_2} = b_{\varphi_1} \& (b_{\varphi_1} \lll 2), \\ &\quad b_{\psi_3} = b_{\psi_2} \mid (b_{\varphi_2} \& (b_{\psi_2} \lll 4)), \quad b_{\varphi_3} = b_{\varphi_2} \& (b_{\varphi_2} \lll 4), \\ &\quad \dots \\ &\quad b_{\psi_k} = b_{\psi_{k-1}} \mid (b_{\varphi_{k-1}} \& (b_{\psi_{k-1}} \lll 2^{k-1})) \text{ in } b_{\psi_k} \end{aligned}$$

Example 2. Consider a trace π over $\mathcal{AP} = \{p, q\}$ as follows:

$$\pi = [\{\}, \{\}, \{p, q\}, \{\}, \{q\}, \{q\}, \{p, q\}].$$

Now consider the MLTL formulas $p, q, \neg p$, and $p \wedge q$. We can encode each as a bit-vector using Definition 5 as follows:

$$\begin{aligned} bv(p, \pi) &= 0b0010001 & bv(q, \pi) &= 0b0010111 \\ bv(\neg p, \pi) &= 0b1101110 & bv(p \wedge q, \pi) &= 0b0010001 \end{aligned}$$

Each bit of each bit-vector represents the truth value of its formula at each timestamp of the trace. For example, $bv(p \wedge q, \pi)[2] = 1$ which corresponds to $\pi, 2 \models p \wedge q$, where p and q both occur at $\pi[2]$.

Example 3. Consider the trace from Example 2 and the MLTL formula $\Diamond_{[1,3]}p$. After first applying $d(\Diamond_{[1,3]}p) = \Diamond_{[1,2]}\Diamond_{[0,1]}p$, we can encode the trace/formula as a bit-vector as follows:

$$\begin{aligned} bv(p, \pi) &= 0b0010001 \\ bv(\Diamond_{[0,1]}p, \pi) &= 0b0110011 \\ bv(\Diamond_{[1,2]}\Diamond_{[0,1]}p, \pi) &= 0b1101110 \end{aligned}$$

Similarly, consider the same trace and the MLTL formula $q \mathcal{U}_{[0,2]}p$ where $d(q \mathcal{U}_{[0,2]}p) = q \mathcal{U}_{[0,1]}(q \mathcal{U}_{[0,1]}p)$:

$$\begin{aligned} bv(p, \pi) &= 0b0010001 \\ bv(q, \pi) &= 0b0010111 \\ bv(q \mathcal{U}_{[0,1]}p, \pi) &= 0b0010011 \\ bv(q \mathcal{U}_{[0,1]}(q \mathcal{U}_{[0,1]}p), \pi) &= 0b0010111 \end{aligned}$$

We now show that the encoding captures the MLTL semantics, where each bit of the resulting bit-vector for an input formula corresponds to whether the input trace satisfies the formula at the index of that bit.

Theorem 1 (MLTL-BV Correctness). *Let $\mathcal{AP}$ be a finite set of atomic propositions, φ be an MLTL_2 formula over $\mathcal{AP}$, and π be a trace. Then for each $\tau < |\pi|$, the bit at index τ of $bv(\varphi, \pi)$ is 1 if and only if π at τ models φ , i.e.,*

$$bv(\varphi, \pi)[\tau] = 1 \text{ iff } (\pi, \tau \models \varphi).$$

Proof. We proceed with induction over the structure of φ . For the following, let π be a finite trace and τ be an index such that $\tau < |\pi|$.

In the base case of our structural induction, $\varphi = p$ where $p \in \mathcal{AP}$. By construction of $bv(p, \pi)$, we have that $bv(p, \pi)[\tau] = 1$ iff $p \in \pi[\tau]$, proving the property.

If $\varphi = \neg\psi$ or $\varphi = \psi \wedge \xi$, then the property holds from the correspondence between $\sim$, $\neg$ and $\&$, $\wedge$.

Now let $\varphi = \Diamond_{[l,u]}\psi$ where $u = l + 2^k - 1$. We inductively assume that the property holds for ψ . Fixing $l \geq 0$, we show that for all $k \geq 0$ it holds that

$$bv(\Diamond_{[l,u]}\psi, \pi)[\tau] = 1 \Leftrightarrow (\pi, \tau \models \Diamond_{[l,u]}\psi).$$

For the base case, $k = 0$ so that $u = l + 2^0 - 1 = l$ and $bv(\Diamond_{[l,l]}\psi, \pi) = bv(\psi, \pi) \ll l$. If $l \geq |\pi|$, then $\pi, \tau \not\models \Diamond_{[l,l]}\psi$ by the MLTL semantics since l is off the end of the trace. By the definition of $\ll$, we also have $bv(\Diamond_{[l,l]}\psi, \pi) = \text{zeroes}(|\pi|)$ so that $bv(\Diamond_{[l,l]}\psi, \pi)[\tau] = 0$, proving the case when $l \geq |\pi|$. Otherwise, we have

$$\begin{aligned} bv(\Diamond_{[l,l]}\psi, \pi)[\tau] &= (bv(\psi, \pi) \ll l)[\tau] \\ &= bv(\psi, \pi)[\tau + l] \end{aligned}$$

which is 1 only if $\pi, \tau + l \models \psi$ by the MLTL semantics and inductive hypothesis, proving the case when $l < |\pi|$.

For the inductive step, assume that the property holds for k , we show that it holds for $k + 1$. Let $b_k = bv(\Diamond_{[l, l+2^k-1]}\psi, \pi)$. Then by Definition 5:

$$\begin{aligned} bv(\Diamond_{[l, l+2^{k+1}-1]}\psi, \pi)[\tau] &= (b_k \mid (b_k \ll 2^k))[\tau] \\ &= b_k[\tau] \vee (b_k \ll 2^k)[\tau] \\ &= b_k[\tau] \vee b_k[\tau + 2^k]. \end{aligned}$$

Since the property holds for k , this value is 1 if and only if

$$\begin{aligned} &\exists i \in [l, l + 2^k - 1]. \pi, (i + \tau) \models \psi \text{ or } \pi, (i + 2^k + \tau) \models \psi \\ \Leftrightarrow &\exists i \in [l, l + 2^k - 1 + 2^k]. \pi, (i + \tau) \models \psi \text{ or } \pi, (i + \tau) \models \psi \\ \Leftrightarrow &\exists i \in [l, l + 2^{k+1} - 1]. \pi, (i + \tau) \models \psi \\ \Leftrightarrow &\pi, \tau \models \Diamond_{[l, l+2^{k+1}-1]}\psi \end{aligned}$$

proving the property for $k + 1$ and completing the case for $\varphi = \Diamond_{[l,u]}\psi$. The proof for the $\mathcal{U}$ case follows the same approach, see Appendix A for the full proof. $\square$

Theorem 2 (MLTL-BV Complexity). *Let $\mathcal{AP}$ be a finite set of atomic propositions, φ be an MLTL_2 formula over $\mathcal{AP}$, K be the size of the largest interval in φ , and π be a trace. Then $bv(\varphi, \pi)$ requires at most $5 \cdot |cl(\varphi)| \cdot \log_2(K)$ bitwise operations to compute.*

Proof. First, let π be a trace and φ be an MLTL formula. Let ψ be the sub-formula of φ with the largest temporal interval size, i.e., the formula associated with $K = \max\{|[l, u]| \mid \psi \in cl_T(\varphi)\}$, and let $[l, u]$ be this interval.

By inspection, if $\varphi = p$, $\varphi = \neg\psi$ or $\varphi = \psi \wedge \xi$, then $bv(\varphi, \pi)$ requires one bitwise operation in addition to the operators of its sub-formulas. Otherwise, if $\varphi = \Diamond_{[l,u]}\psi$, then $bv(\varphi, \pi)$ requires an additional $\log_2(K)$ shifts and $\log_2(K - 1)$ bitwise-or operations totaling $\log_2(K) + \log_2(K - 1) < 2\log_2(K)$ bitwise operations. Alternatively, if $\varphi = \psi \mathcal{U}_{[l,u]}\xi$, then $bv(\varphi, \pi)$ requires $2\log_2(K)$ shifts, $\log_2(K - 1)$ bitwise-or operations, and $2\log_2(K - 1)$ bitwise ands totaling $2\log_2(K) + 3\log_2(K - 1) < 5\log_2(K)$ bitwise operations. Therefore $bv(\varphi, \pi)$ requires no more than $|cl(\varphi)| \cdot 5\log_2(K)$ bitwise operations in total to compute. $\square$

For an MLTL formula φ , computing $bv(d(\varphi), \pi)$ requires at most $\mathcal{O}(|cl(\varphi)| \cdot (\log_2(K))^2)$ operations since $|cl(d(\varphi))| \leq |cl(\varphi)| \cdot \log_2(K)$.

IV. SATISFIABILITY CHECKING VIA MLTL-BV

A. Problem Encoding and Algorithm

We can use the MLTL-BV encoding to solve the satisfiability problem for MLTL, recalling that this poses whether there exists some trace π such that $\pi, 0 \models \varphi$. Using our bit-vector encoding, we can query whether there exists a bit-vector $bv(p)$ for each $p \in \mathcal{AP}$ such that there is at least one bit set in $bv(\varphi)$ (i.e., $\text{redor}(bv(\varphi)) = 1$). We begin by proving a bound on the size of satisfying traces using Corollary 1.

Lemma 4. *An MLTL formula φ over $\mathcal{AP}$ is satisfiable iff there exists a trace π of length $|\pi| \leq \text{cplen}(\varphi)$ such that $\pi, 0 \models \varphi$.*

Proof. ($\Leftarrow$) Assume that there exists a trace π of length $|\pi| \leq \text{cplen}(\varphi)$ such that $\pi, 0 \models \varphi$. Then φ is satisfiable.

($\Rightarrow$) Assume φ is satisfiable and let π be a trace such that $\pi, 0 \models \varphi$. If $|\pi| \leq \text{cplen}(\varphi)$, then the property holds directly. Otherwise, $|\pi| > \text{cplen}(\varphi)$. Then taking the prefix of π with length $\text{cplen}(\varphi)$ defined as $\pi' = \pi[0], \pi[1], \dots, \pi[\text{cplen}(\varphi) - 1]$, we have from Corollary 1 that $\pi', 0 \models \varphi$. Since $|\pi'| = \text{cplen}(\varphi) \leq \text{cplen}(\varphi)$, the property holds. $\square$

Theorem 3. *An MLTL formula φ over $\mathcal{AP}$ is satisfiable iff there exists a trace π with $|\pi| \leq \text{cplen}(\varphi)$ such that $\text{redor}(bv(\varphi, \pi)) = 1$.*

Proof. ($\Rightarrow$) Assume that φ is satisfiable. From Lemma 4, there exists a finite trace π such that $|\pi| \leq \text{cplen}(\varphi)$ and $\pi, 0 \models \varphi$. From Theorem 1, it holds that $bv(\varphi, \pi)[0] = 1$ where the width of $bv(\varphi, \pi)$ is $|\pi| = \text{cplen}(\varphi)$, so $\text{redor}(bv(\varphi, \pi)) = 1$.

($\Leftarrow$) Assume that there exists a trace π with $|\pi| \leq \text{cplen}(\varphi)$ such that $\text{redor}(bv(\varphi, \pi)) = 1$. Let $\tau < |\pi|$ be an index of $bv(\varphi, \pi)$ such that $bv(\varphi, \pi)[\tau] = 1$, of which at least 1 such index exists from the definition of redor . Therefore from Theorem 1 it holds that $\pi, \tau \models \varphi$. $\square$

We use this Theorem to encode the MLTL satisfiability problem into an SMTLIB2 [31] query using the theory of quantifier-free bit-vectors (QF_BV). Specifically, given an MLTL formula φ , we declare an uninterpreted bit-vector for each proposition of width equal to $\text{cplen}(\varphi)$ (representing a trace π), apply the MLTL-BV encoding to φ by defining

```

1 (declare-fun |p| () (_ BitVec 4)) (declare-fun |q| () (_ BitVec 4))
2 (define-fun b_0 () (_ BitVec 4) (bvshl |q| (_ bv1 4)))
3 (define-fun |F[0,1] |q| () (_ BitVec 4) (bvor b_0 |q|))
4 (define-fun b_1 () (_ BitVec 4) (bvshl |F[0,1] |q| (_ bv3 4)))
5 (define-fun b_2 () (_ BitVec 4) (bvshl b_1 (_ bv1 4)))
6 (define-fun |F[3,4] |F[0,1] |q| () (_ BitVec 4) (bvor b_1 b_2))
7 (define-fun |p & F[3,4] |F[0,1] |q| () (_ BitVec 4) (bvand |p| |F[1,2] |F[0,1] |q|))
8 (assert (bvugt |p & F[3,5] |q| (_ bv0 4)))

```

Fig. 1: SMTLIB2 encoding for the formula $p \wedge \Diamond_{[3,5]} q$. The encoding uses two uninterpreted bit-vectors for p and q (line 1) and then Definition 5 for the operators (lines 2-7). The assertion checks whether there exist bit-vectors for p and q such that the top-level operator of the formula’s bit-vector value could be greater than zero (line 8).

a function for each operator in φ as in Definition 5, then assert that the corresponding function for φ is greater than zero (this is the same as `redor`, since `redor` is unsupported in the QF_BV logic). Figure 1 shows an example SMTLIB2 encoding for the formula $p \wedge \Diamond_{[1,3]} q$.

Incremental Checking We also contribute an incremental version of this encoding. Notice that the above encoding tries to find a trace of length $cplen(\varphi)$ or less in one shot; in principle we can query whether a trace of any arbitrary length or less satisfies φ by setting the bit-vector width accordingly. Many MLTL formulas are satisfied by much shorter traces than $cplen(\varphi)$. For example, according to the MLTL semantics (Definition 2), the formula $\neg(\Diamond_{[2,5]} p) \equiv \Box_{[2,5]} \neg p$ is satisfied by any trace of length 1, since p technically never holds in the range $[2, 5]$. To leverage this, we first check if bit-vectors of width 1 satisfy the formula φ , then of width equal to φ ’s best-case propagation delay [4] (intuitively, the minimal sum of the formula’s lower bounds), then finally of width $cplen(\varphi)$. We denote this approach as QF_BV *.

B. Experimental Evaluation

We evaluate the encoding, comparing against existing state-of-the-art MLTL satisfiability-checking techniques, specifically the approaches presented in [9]. That work investigated reduction of MLTL satisfiability checking to LTL_f satisfiability checking, model checking, and SMT solving, where the SMT encoding outperformed the other approaches in general. Their primary SMT encoding uses the theory of uninterpreted functions and linear integer arithmetic (UFLIA). The authors also investigated a quantifier-free encoding (QF_UFLIA), but found it did not scale as well as interval size increased. We therefore focus our comparison to the UFLIA encoding.

Experimental Goals. Our experiments aim to answer the following research questions: (RQ1) Can the QF_BV encoding improve MLTL satisfiability checking performance compared to the UFLIA encoding? (RQ2) How does the performance of the QF_BV and QF_BV * approaches vary with the maximum interval size in MLTL formulas? (RQ3) Is there a performance advantage in using an incremental approach (QF_BV *) for large-interval formulas?

Tools and Environment. For the UFLIA encoding, we use the solvers Z3 [32], CVC5 [33], and for the QF_BV encodings YICES2 [34] and BITWUZLA [35], taking the smaller time from either solver for each encoding. Experimentation showed

Max Bnd	Encoding	Solved	Sat	Unsat	Best	Unique
10	UFLIA	9,970	9,638	332	2	0
	QF_BV	10,000	9,668	332	9,998	30
100	UFLIA	9,774	9,473	301	7,512	0
	QF_BV	10,000	9,696	304	2,488	226
1,000	UFLIA	9,538	9,249	289	3,535	3
	QF_BV	9,994	9,704	290	600	0
	QF_BV *	9,994	9,704	290	5,862	0
10,000	UFLIA	9,299	8,981	318	3,885	452
	QF_BV	7,953	7,700	253	389	0
	QF_BV *	9,251	8,986	265	5,430	31

TABLE I: Evaluation results showing number of instances solved (Solved), satisfiable solved (Sat), unsatisfiable solved (Unsat), fastest time to solve across all encodings (Best), and instances solved by only that encoding (Unique). The highest value in each benchmark set for each statistic is bolded.

that BITWUZLA and YICES2 outperformed both CVC5 and Z3 on virtually all QF_BV instances, and BITWUZLA and YICES2 do not support the UFLIA theory. We ran experiments on a cluster with 2x 32-core AMD EPYC 7502 processors and 500GB of RAM, with a timeout of 20 minutes and memout of 16GB per instance. We report only solving time; encoding time for all instances was less than 5 seconds.

Benchmarks. We use pseudo-randomly-generated MLTL formulas via the same technique as in [9]: we generate 4 sets of 10,000 formulas with maximum temporal upper bounds in the range 10, 100, 1,000, and 10,000. Each set is composed of 100 formula subsets each with varying number of propositions (1 to 5), formula closure size (20, 40, 60, 80, and 100), and probability of each operator in the formula being temporal (33%, 50%, 75%, and 95%). We investigated other benchmarks, including formulas derived from case studies [9], [36], [37], and found similar results to the random formulas with upper bounds of 10,000. We also only employ the QF_BV * approach when the interval size is 1,000 or greater since the QF_BV approach solves all problems with sizes 10 and 100.

Experimental Results. Comparing performance across the approaches (RQ1), Table I summarizes results across all four sets of benchmarks and Figure 2 shows cactus plots for the 1,000 and 10,000 interval benchmarks. Across all benchmarks, the QF_BV-based approaches solved 743 instances that the UFLIA encoding could not and was faster than UFLIA on 62% of instances, demonstrating its effectiveness over the UFLIA in many instances.

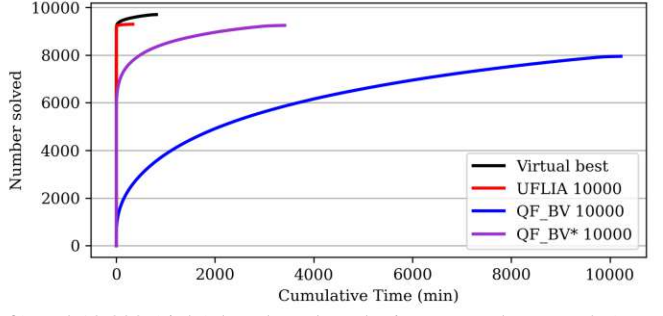
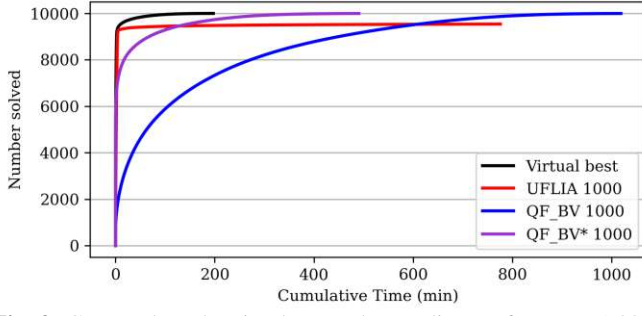


Fig. 2: Cactus plots showing how each encoding performs on 1,000 (left) and 10,000 (right) benchmarks. The incremental approach (purple) outperforms the standard approach (blue), and is complementary to the UFLIA encoding (red) as the virtual best (black) is better than any one encoding alone.

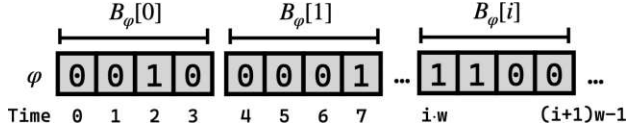


Fig. 3: Visualizing the contents of a buffer B_φ up to word i with a word size $w = 4$. At word $B_\varphi[i]$, the first bit represents φ 's value at $i * w$ and the final bit at $(i + 1) * w - 1$.

To study how performance varies with interval size (RQ2), we observe that for smaller intervals (10 and 100), the QF_BV encoding consistently outperforms the UFLIA baseline. For larger intervals (1,000 and 10,000), the incremental QF_BV* encoding shows the best performance on the 1,000 benchmark set and remains highly competitive on the 10,000 set.

Regarding the benefit of incremental reasoning (RQ3), the QF_BV* variant clearly improves upon the non-incremental QF_BV encoding at large interval sizes. On average, the incremental approach required one query for 50.5% of formulas, two queries for 20.8% of formulas, and a full three queries for the remaining 28.7%, indicating that incremental strategies are particularly valuable when dealing with such formulas.

V. MONITORING

Given a finite trace π and an MTTL formula φ , the offline monitoring problem consists of computing whether $\pi, \tau \models \varphi$ holds for every time point τ . Conveniently, Definition 5 offers a direct method for solving this problem. In this section, we turn our attention to the online monitoring problem, which is similar in principle but differs in that π is treated as an unbounded stream of states.

In adapting our encoding for online monitoring, our algorithm aims to satisfy three key criteria: takes a stream of states as input and produces a stream of verdicts as output, operates exclusively on fixed-size bit-vectors, and avoids redundant computation by never re-evaluating parts of any bit-vector. Hardware considerations motivate the focus on fixed-size bit-vectors; most machines natively support operations on bit-vectors of size 8, 16, 32, or 64 bits. We refer to these fixed-size bit-vectors as *words* and assume a fixed word size w .

Given an input formula φ , the monitor consumes states from the input stream and stores them in a buffer, interpreting this buffer as a trace π . Then, for each sub-formula $\psi \in cl(\varphi)$, the monitor maintains a buffer B_ψ of words, i.e., $B_\psi[i]$ is a word

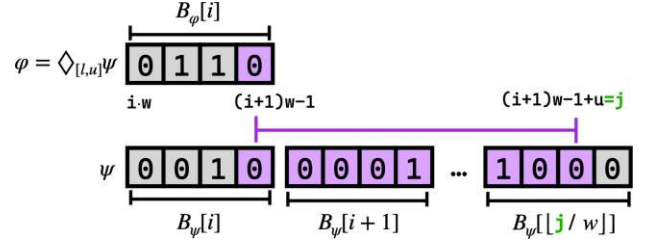


Fig. 4: The final bit of $B_\varphi[i]$, representing φ 's value at time $(i + 1) * w - 1$, relies on values of ψ through time $j = (i + 1) * w - 1 + u$ (purple), which is stored in word $B_\psi[\lfloor j/w \rfloor]$. The monitor therefore waits until $B_\psi[\lfloor j/w \rfloor]$ is fully computed before computing $B_\varphi[i]$.

and $B_\psi[i][j]$ (with $j < w$) is a bit representing the value of ψ at time $i * w + j$ (Figure 3). For now, we assume the size of these buffers is unbounded.

The monitor computes the words of these buffers one at a time via Definition 5. Intuitively, when the monitor performs the computation of $bv(\psi, \pi)$, it stores the relevant bits of the result in $B_\psi[i]$ and uses B_ψ as a cache whenever referencing $bv(\psi, \pi)$ in subsequent computations. For instance, the monitor for $\psi = \neg\xi$ will compute $B_\psi[i] = \sim B_\xi[i]$, using $B_\xi[i]$ in place of $bv(\xi, \pi)$.

For temporal operators, the monitor waits until all necessary information is available from its direct sub-formulas before computing via Definition 5. Consider that to compute the value for $\psi = \Diamond_{[l,u]}\xi$ at time τ requires the values of ξ through time $\tau + u$ (in the worst case). Now, computing the full word $B_\psi[i]$ means computing ψ 's values for times $i * w$ through $(i + 1) * w - 1$, therefore requiring values for ξ through time $j = ((i + 1) * w - 1) + u$ (see Figure 4). This value is stored in the word $B_\xi[\lfloor j/w \rfloor]$, so the monitor waits until this word is computed before computing $B_\psi[i]$. The same idea applies for the $\mathcal{U}_{[l,u]}$ operator.

Up to now we have assumed the buffers are unbounded, but we can drop values of the buffers once we no longer need them. For example, given an input formula φ , once the monitor computes $B_\varphi[0]$, it no longer needs the values for $B_p[0]$. We introduce a value $W(\psi, w)$ for an MTTL₂ formula ψ and word size $w > 0$ that represents the number of words ahead each

Spec Name	MLTL Formula
Future	$\Diamond_{[0,10]} p$
Until	$p \mathcal{U}_{[0,10]} q$
Minimum Duration	$p \vee ((\neg p \mathcal{U}_{[0,10]} p) \vee \Box_{[0,10]} \neg p)$
Eventually p Between q and r	$(q \wedge \Box_{[0,10]} \neg r \wedge \Diamond_{[10,30]} r) \rightarrow ((\neg r \mathcal{U}_{[10,20]} (p \wedge \neg r)) \vee \Box_{[10,20]} \neg p)$
Precedence Chain	$\Diamond_{[10,10]} (q_0 \wedge \Diamond_{[0,5]} (q_1 \wedge \Diamond_{[0,5]} q_2 \wedge (\Diamond_{[0,5]} q_3))) \rightarrow \Diamond_{[0,10]} p$

TABLE II: The instantiated pattern formulas [26] used for benchmarking provide some simple formulas (Future, Until), a larger formula over a single proposition (Min Duration), a complex formula capturing temporal ordering of 3 propositions (Between), and a formula featuring nesting of temporal operators (Precedence Chain). sub-formula needs to look to compute the current word:

$$\begin{aligned}
W(p, w) &= 1, \quad W(\neg \varphi, w) = W(\varphi, w) \\
W(\varphi \wedge \psi, w) &= \max(W(\varphi, w), W(\psi, w)) \\
W(\Diamond_{[l,u]} \psi, w) &= \lfloor ((w-1) + u)/w \rfloor + W(\psi, w) \\
W(\varphi \mathcal{U}_{[l,u]} \psi, w) &= \lfloor ((w-1) + u)/w \rfloor + \max(W(\varphi), W(\psi))
\end{aligned}$$

Here, $(w-1) + u$ represents how many bits ahead the monitor needs to compute the final bit of the current word. Then, given an input formula φ , each buffer $\mathbf{B}_\psi$ for $\psi \in cl(\varphi)$ is assigned the size $|\mathbf{B}_\psi| = W(\varphi, w)$. Therefore the size of φ 's monitor is bounded by $|cl(\varphi)| \cdot W(\varphi, w)$ words. As an example:

$$\begin{aligned}
W(\Diamond_{[0,1]} \Diamond_{[0,1]} p, 4) &= \lfloor (3+1)/4 \rfloor + W(\Diamond_{[0,1]} p, 4) \\
&= 1 + \lfloor (3+1)/4 \rfloor + 1 = 3
\end{aligned}$$

We note this is not a tight bound (e.g., the input formula φ only strictly requires a buffer size of 1 to store the current word). Proving a tighter bound is left for future work.

Due to this waiting, the algorithm cannot compute the value of φ at time τ until consuming $W(\varphi, w) * w$ states beyond τ . This is a significant drawback as the monitor cannot report violations of the formula as soon as they occur. This is a major focus for future work as this is often a requirement for time-sensitive, safety-critical systems.

A. Experimental Evaluation

We implemented a prototype tool, SABRE (Stream Analysis via Bitwise Reasoning), that takes an MLTL formula as input and produces a C program that monitors the formula according to the presented algorithm. The C program takes as input a stream of states (in the form of a CSV file read from standard input) and outputs a stream of verdicts.

Experimental Goals. Our experiments aim to answer the following research questions: (RQ1) How does SABRE perform in terms of throughput and memory usage compared to existing tools when applied to realistic specifications? (RQ2) How does its performance vary with the shape of the input trace, particularly with respect to the frequency with which propositions occur? (RQ3) How does SABRE respond to different interval sizes, especially when those sizes are exact powers of two?

Tools and Environment. We compare SABRE to two other tools. The first tool we compare to is R2U2 [24], an online MLTL monitoring framework designed for resource-constrained real-time embedded systems. R2U2 offers a

	Maximum % Throughput Difference			
	$\Diamond_{[0,1000]} p$	$\Diamond_{[0,1023]} p$	$\Diamond_{[0,5000]} p$	$\Diamond_{[0,10000]} p$
R2U2	37.20%	38.69%	38.19%	38.03%
HYDRA	11.58%	13.29%	11.59%	12.23%
SABRE	4.00%	8.33%	5.71%	4.00%

TABLE III: Percent differences between the highest and lowest throughput across all trace densities demonstrate that SABRE is the most robust to changes in trace density, offering relatively stable performance no matter the shape of the trace.

Rust [38] and C implementation. For our experiments, we use the C version, although both versions offered similar performance.

The second tool we compare against is HYDRA [17], an offline MTL monitoring tool. Since HYDRA monitors MTL formulas, we convert all MLTL formulas of the form $\varphi \mathcal{U}_{[l,u]} \psi$ to the equivalent MTL formula $\Diamond_{[l,l]} (\varphi \mathcal{U}_{[0,u-l]} \psi)$. HYDRA also takes as input a sequence of timestamped events instead of the traces as defined above. We convert each trace to a HYDRA-compatible trace by reporting an event for every timestamp in the trace regardless of whether the event is empty or not. In comparing SABRE to HYDRA we aim to gain insight as to how our method compares to an algorithm intended for offline usage. This also implicitly compares our method to other tools including MONPOLY [14], DEJAVU [15], AERIAL [39], and REELAY [40] as HYDRA outperformed those tools in experimentation [17], [41].

Each experiment involved running a tool on a specific formula/trace pair. We measured the throughput, defined as the number of verdicts reported divided by the total execution time, and the peak RAM usage, reported as the maximum resident set size (RSS) using the GNU `time` command. To ensure consistency, we sent all tool output to `/dev/null` and executed each formula/trace pair 10 times, reporting the median execution time and max RSS across those runs to reduce the impact of outliers, e.g., due to loading of shared libraries. Execution time does not include encoding or compilation time, though in all cases this time was less than 1 second and would not impact performance in an online setting. All experiments were conducted on a machine with an Apple M3 Pro processor and 18GM RAM.

Benchmarks. We use two sets of benchmarks to exercise the algorithm. **P** is a set of five formulas derived from a case study on common Patterns of MTL formulas [26], shown in Table II with interval sizes up to 20. **F** (for **Future**) is the formula $\Diamond_{[0,u]} p$ with the values $u = 1000, 1023, 5000$ and 10000.

We run all experiments over pseudo-random traces of length 5,000,000; long enough so that each tool's runtime is reliably measurable by the `time` command (i.e., on the order of seconds). We generated each trace with a specified *density*, defined as the relative proportion of 1s to 0s on average for each proposition in the trace. For **P**, all traces have a density of 0.5, so each proposition was present half as often as it was absent on average. For **F**, we vary the density to see its effect on each tool's performance.

Experimental Results. When applied to the **P** formulas

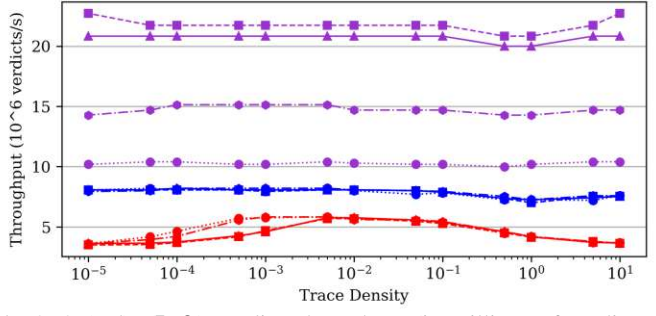
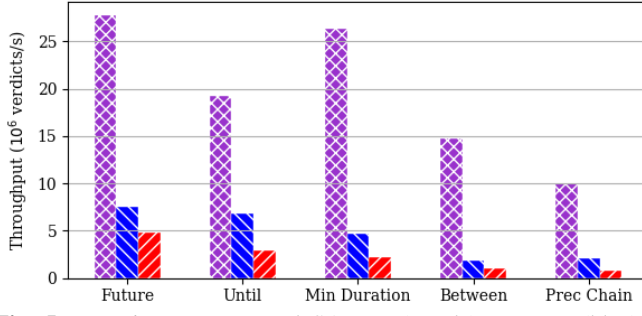


Fig. 5: Experiments compared SABRE (purple), HYDRA (blue), and R2U2 (red). **(Left)** Median throughput (in millions of verdicts per second) for each formula in Table II shows that SABRE performs best across all formulas. **(Right)** Varying trace density for $\diamond_{[0,1000]}p$ (solid, ▲), $\diamond_{[0,1023]}p$ (dashed, ■), $\diamond_{[0,5000]}p$ (dash-dotted, ●), $\diamond_{[0,10000]}p$ (dotted, ●). All HYDRA lines coincide.

(RQ1), SABRE consistently outperformed both R2U2 and HYDRA in terms of throughput (Figure 5, left), achieving an average of **10.52x** higher throughput than R2U2 on the P formulas and **4.96x** higher than HYDRA, despite HYDRA being an offline tool. On average, HYDRA’s max RSS was **56.79MB** while R2U2’s was **1.09MB** and SABRE’s **1.05MB**, suggesting SABRE is not using more memory as a tradeoff to achieve its throughput.

Figure 5 (right) shows how each tool performs across a range of densities where it appears that HYDRA is the tool most insensitive to trace shape. However, since SABRE’s throughput is higher than HYDRA’s, we consider the percent difference between the maximum and minimum throughput for each tool on each formula (Table III). Taking the formula $\diamond_{[0,1023]}p$, for which SABRE has the largest such percent difference, R2U2’s maximum throughput is **37.20%** percent higher than its minimum across all densities, HYDRA’s is **11.58%** higher, and SABRE’s is only **8.33%** higher, demonstrating that SABRE is more robust with respect to trace shape than R2U2 or HYDRA. This is likely due to the algorithm not including any branching, i.e., always performing identical bit-vector operations at each execution no matter the values of the input.

SABRE’s performance varied substantially with interval size (RQ3). It performed better on the formula $\diamond_{[0,1023]}p$ than $\diamond_{[0,1000]}p$, despite the former having a larger interval, highlighting the advantage of the encoding when interval sizes are powers of two. Performance tapered at larger intervals (e.g., 10000), approaching that of HYDRA, whereas the other tools do not suffer from such a performance degradation (Figure 5, right). This is likely due to the other tools’ methods allowing early evaluation of temporal operators and so may only use some information in the interval, whereas SABRE uses all information in the temporal interval regardless.

VI. CONCLUSION

There are several promising directions for future work, especially in the context of monitoring. First, the algorithm’s structure lends itself naturally to parallelization, as many sub-formulas are independent and can be evaluated concurrently. This is especially true in the offline monitoring setting, where the entire trace is available in advance, allowing evaluations

to begin from multiple points in the trace simultaneously. Additionally, the encoding could possibly be adapted to use matrix operations, enabling the use of optimized GPU-based tools to further accelerate performance. Beyond parallelization, we also plan to extend the encoding framework to support other temporal logics, including both finite-trace variants and infinite-trace logics by encoding lasso-shaped traces. There is also potential for applying these techniques to non-boolean inputs.

Our findings suggest that reducing problems to bit-vector operations, as temporal exponentiation does so effectively, is a promising approach to problems related to temporal logic reasoning.

REFERENCES

- [1] K. Y. Rozier, “Specification: The biggest bottleneck in formal methods and autonomy,” in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2016, pp. 8–26.
- [2] J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel, “Scalable online first-order monitoring,” *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 2, pp. 185–208, 2021.
- [3] T. Reinbacher, K. Y. Rozier, and J. Schumann, “Temporal-logic based runtime observer pairs for system health management of real-time systems,” in *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science (LNCS), vol. 8413. Springer-Verlag, April 2014, pp. 357–372.
- [4] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier, “Embedding online runtime verification for fault disambiguation on robonaut2,” in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2020, pp. 196–214.
- [5] A. Aurandt, P. Jones, and K. Y. Rozier, “Runtime Verification Triggers Real-time, Autonomous Fault Recovery on the CySat-I,” in *Proceedings of the 14th NASA Formal Methods Symposium (NFM 2022)*, ser. Lecture Notes in Computer Science (LNCS), vol. 13260. Caltech, California, USA: Springer, Cham, May 2022.
- [6] A. Hammer, M. Cauwels, B. Hertz, P. Jones, and K. Y. Rozier, “Integrating runtime verification into an automated uas traffic management system,” *Innovations in Systems and Software Engineering: A NASA Journal*, July 2021.
- [7] C. Johannsen, M. Anderson, W. Burken, E. Diersen, J. Edgren, C. Glick, S. Jou, A. Kumar, J. Levandowski, E. Moyer, T. Roquet, A. VandeLoo, and K. Y. Rozier, *OpenUAS Version 1.0*. Athens, Greece (Virtual): IEEE, June 2021.
- [8] J. B. Dabney, J. M. Badger, and P. Rajagopal, “Adding a verification view for an autonomous real-time system architecture,” in *Proceedings of SciTech Forum*, ser. 2021-0566. AIAA, January 2021, p. Online.
- [9] J. Li, M. Y. Vardi, and K. Y. Rozier, “Satisfiability checking for mission-time ltl (mltl),” *Information and Computation*, vol. 289, p. 104923, 2022.
- [10] R. Koymans, “Specifying real-time properties with metric temporal logic,” *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

- [11] D. Basin, F. Klaedtke, and E. Zălinescu, “Algorithms for monitoring real-time properties,” in *International Conference on Runtime Verification*. Springer, 2011, pp. 260–275.
- [12] D. Basin, B. N. Bhatt, S. Krstić, and D. Traytel, “Almost event-rate independent monitoring,” *Formal Methods in System Design*, vol. 54, pp. 449–478, 2019.
- [13] D. Basin, F. Klaedtke, and S. Müller, “Monitoring security policies with metric first-order temporal logic,” in *Proceedings of the 15th ACM symposium on Access control models and technologies*, 2010, pp. 23–34.
- [14] D. Basin, F. Klaedtke, and E. Zălinescu, “The monpoly monitoring tool,” in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, vol. 3. EasyChair, 2017, pp. 19–28.
- [15] K. Havelund, D. Peled, and D. Ulus, “Dejavu: A monitoring tool for first-order temporal logic,” in *2018 IEEE Workshop on Monitoring and Testing of Cyber-Physical Systems (MT-CPS)*, 2018, pp. 12–13.
- [16] D. A. Basin, S. Krstić, and D. Traytel, “Aerial: Almost event-rate independent algorithms for monitoring metric regular properties,” in *RV-CuBES*, 2017, pp. 29–36.
- [17] M. Raszkyk, D. Basin, S. Krstić, and D. Traytel, “Multi-head monitoring of metric temporal logic,” in *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 151–170.
- [18] B. d’Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, “Lola: runtime monitoring of synchronous systems,” in *12th International Symposium on Temporal Representation and Reasoning (TIME’05)*. IEEE, 2005, pp. 166–174.
- [19] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah, “Streamlab: stream-based monitoring of cyber-physical systems,” in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 421–431.
- [20] I. Perez, F. Dedden, and A. Goodloe, “Copilot 3,” Tech. Rep., 2020.
- [21] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, “Tessla: temporal stream-based specification language,” in *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*. Springer, 2018, pp. 144–162.
- [22] J. Schumann, P. Moosbrugger, and K. Y. Rozier, “R2u2: monitoring and diagnosis of security threats for unmanned aerial systems,” in *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22–25, 2015. Proceedings*. Springer, 2015, pp. 233–249.
- [23] K. Y. Rozier and J. Schumann, “R2u2: tool overview,” in *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, no. ARC-E-DAA-TN48849, 2017.
- [24] C. Johannsen, P. Jones, B. Kempa, K. Y. Rozier, and P. Zhang, “R2u2 version 3.0: Re-imagining a toolchain for specification, resource estimation, and optimized observer generation for runtime verification in hardware and software,” in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 483–497.
- [25] M. Valizadeh, N. Fijalkow, and M. Berger, “Ltl learning on gpus,” in *International Conference on Computer Aided Verification*. Springer, 2024, pp. 209–231.
- [26] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, “Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar,” *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.
- [27] C. Johannsen, P. Jones, K. Y. Rozier, and T. Wongpiromsarn, “MLTL Reasoning via Bit-vector Encoding, FMCAD 2025 Artifact,” Jul. 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.16312254>
- [28] J. Elwing, L. Gamboa-Guzman, J. Sorkin, C. Travesset, Z. Wang, and K. Y. Rozier, “Mission-time ltl (mltl) formula validation via regular expressions,” in *International Conference on Integrated Formal Methods*. Springer, 2023, pp. 279–301.
- [29] K. Kosaian, Z. Wang, E. Sloan, and K. Rozier, “Formalizing mltl formula progression in isabelle/hol,” *arXiv preprint arXiv:2410.03465*, 2024.
- [30] C. Johannsen, B. Kempa, P. H. Jones, K. Y. Rozier, and T. Wongpiromsarn, “Impossible made possible: Encoding intractable specifications via implied domain constraints,” in *Formal Methods for Industrial Critical Systems*, A. Cimatti and L. Titolo, Eds. Cham: Springer Nature Switzerland, 2023, pp. 151–169.
- [31] 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)*, vol. 13, 2010, p. 14.
- [32] 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*. Springer, 2008, pp. 337–340.
- [33] 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 *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 415–442.
- [34] B. Dutertre, “Yices 2.2,” in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 737–744.
- [35] A. Niemetz and M. Preiner, “Bitwuzla,” in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 3–17.
- [36] M. Gario, A. Cimatti, C. Mattarei, S. Tonetta, and K. Y. Rozier, “Model checking at scale: Automated air traffic control design space exploration,” in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part II 28*. Springer, 2016, pp. 3–22.
- [37] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, “Formal design and safety analysis of air6110 wheel brake system,” in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I 27*. Springer, 2015, pp. 518–535.
- [38] A. Aurandt, P. H. Jones, and K. Y. Rozier, “Towards a safe, verified runtime monitor for embedded systems: R2u2 in embedded rust,” in *Proceedings of the 17th NASA Formal Methods Symposium (NFM 2025)*, 2025.
- [39] D. Basin, B. N. Bhatt, and D. Traytel, “Almost event-rate independent monitoring of metric temporal logic,” in *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Part II 23*. Springer, 2017, pp. 94–112.
- [40] D. Ulus, “Reelay,” <https://github.com/doganulus/reelay>, 2025.
- [41] M. Raszkyk, D. Basin, and D. Traytel, “Multi-head monitoring of metric dynamic logic,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 233–250.

Theorem 1 (MLTL-BV Correctness). *Let $\mathcal{AP}$ be a finite set of atomic propositions, φ be an MLTL_2 formula over $\mathcal{AP}$, and π be a trace. Then for each $\tau < |\pi|$, the bit at index τ of $bv(\varphi, \pi)$ is 1 if and only if π at τ models φ , i.e.,*

$$bv(\varphi, \pi)[\tau] = 1 \text{ iff } (\pi, \tau \models \varphi).$$

Proof. We proceed with induction over the structure of φ . For the following, let π be a finite trace and τ be an index such that $\tau < |\pi|$.

In the base case of our structural induction, $\varphi = p$ where $p \in \mathcal{AP}$. By construction of $bv(p, \pi)$, we have that $bv(p, \pi)[\tau] = 1$ iff $p \in \pi[\tau]$, proving the property.

If $\varphi = \neg\psi$ or $\varphi = \psi \wedge \xi$, then the property holds from the correspondence between $\sim$, $\neg$ and $\&$, $\wedge$.

Now let $\varphi = \Diamond_{[l, u]}\psi$ where $u = l + 2^k - 1$. We inductively assume that the property holds for ψ . Fixing $l \geq 0$, we show that for all $k \geq 0$ it holds that

$$bv(\Diamond_{[l, u]}\psi, \pi)[\tau] = 1 \Leftrightarrow (\pi, \tau \models \Diamond_{[l, u]}\psi).$$

For the base case, $k = 0$ so that $u = l + 2^0 - 1 = l$ and $bv(\Diamond_{[l, l]}\psi, \pi) = bv(\psi, \pi) \ll l$. If $l \geq |\pi|$, then $\pi, \tau \not\models \Diamond_{[l, l]}\psi$ by the MLTL semantics since a is off the end of the trace. By the definition of $\ll$, we also have $bv(\Diamond_{[l, l]}\psi, \pi) = \text{zeroes}(|\pi|)$ so that $bv(\Diamond_{[l, l]}\psi, \pi)[\tau] = 0$, proving the case when $l \geq |\pi|$. Otherwise, we have

$$\begin{aligned} bv(\Diamond_{[l, l]}\psi, \pi)[\tau] &= (bv(\psi, \pi) \ll l)[\tau] \\ &= bv(\psi, \pi)[\tau + l] \end{aligned}$$

which is 1 only if $\pi, \tau + l \models \psi$ by the MLTL semantics and inductive hypothesis, proving the case when $l < |\pi|$.

For the inductive step, assume that the property holds for k , we show that it holds for $k + 1$. Let $b_k = bv(\Diamond_{[l, l+2^k-1]}\psi, \pi)$. Then by Definition 5:

$$\begin{aligned} bv(\Diamond_{[l, l+2^{k+1}-1]}\psi, \pi)[\tau] &= (b_k \mid (b_k \ll 2^k))[\tau] \\ &= b_k[\tau] \vee (b_k \ll 2^k)[\tau] \\ &= b_k[\tau] \vee b_k[\tau + 2^k]. \end{aligned}$$

Since the property holds for k , this value is 1 if and only if

$$\begin{aligned} &\exists i \in [l, l + 2^k - 1]. \pi, (i + \tau) \models \psi \text{ or } \pi, (i + 2^k + \tau) \models \psi \\ \Leftrightarrow &\exists i \in [l, l + 2^k - 1 + 2^k]. \pi, (i + \tau) \models \psi \text{ or } \pi, (i + \tau) \models \psi \\ \Leftrightarrow &\exists i \in [l, l + 2^{k+1} - 1]. \pi, (i + \tau) \models \psi \\ \Leftrightarrow &\pi, \tau \models \Diamond_{[l, l+2^{k+1}-1]}\psi \end{aligned}$$

proving the property for $k + 1$ and completing the case for $\varphi = \Diamond_{[l, u]}\psi$.

Let $\varphi = \psi \mathcal{U}_{[l, u]}\xi$ where $u = l + 2^k - 1$. We inductively assume that the property holds for ψ and ξ . Fixing $l \geq 0$, we show that for all $k \geq 0$ it holds that

$$bv(\psi \mathcal{U}_{[l, u]}\xi, \pi)[\tau] = 1 \Leftrightarrow (\pi, \tau \models \psi \mathcal{U}_{[l, u]}\xi).$$

For the base case, $k = 0$ so that $u = l + 2^0 - 1 = l$. Then $bv(\psi \mathcal{U}_{[l, l]}\xi, \pi) = bv(\Diamond_{[l, l]}\xi, \pi) = bv(\xi, \pi) \ll l$. If $l \geq |\pi|$, then $\pi, \tau \not\models \Diamond_{[l, l]}\xi$ by the MLTL semantics since a is off the end of the trace. By the definition of $\ll$, we also have $bv(\Diamond_{[l, l]}\xi, \pi) = \text{zeroes}(|\pi|)$ so that $bv(\Diamond_{[l, l]}\xi, \pi)[\tau] =$

$bv(\psi \mathcal{U}_{[l, l]}\xi) = 0$, proving the case when $l \geq |\pi|$. Otherwise, we have

$$\begin{aligned} bv(\psi \mathcal{U}_{[l, l]}\xi, \pi)[\tau] &= bv(\Diamond_{[l, l]}\xi, \pi)[\tau] \\ &= (bv(\xi, \pi) \ll l)[\tau] \\ &= bv(\xi, \pi)[\tau + l] \end{aligned}$$

which is 1 only if $\pi, \tau + l \models \xi$ by the MLTL semantics and inductive hypothesis, proving the case when $l < |\pi|$.

For the inductive step, assume that the property holds for k , we show that it holds for $k + 1$. Let $b_{\psi_k} = bv(\Box_{[l, l+(2^k-1)]}\psi, \pi)$ and $b_{\xi_k} = bv(\psi \mathcal{U}_{[l, l+(2^k-1)]}\xi, \pi)$. Then by Definition 5:

$$\begin{aligned} bv(\psi \mathcal{U}_{[l, l+(2^{k+1}-1)]}\xi, \pi)[\tau] &= (b_{\xi_k} \mid (b_{\psi_k} \& (b_{\xi_k} \ll 2^k)))[\tau] \\ &= b_{\xi_k}[\tau] \vee (b_{\psi_k}[\tau] \wedge b_{\xi_k}[\tau + 2^k]) \end{aligned}$$

by the MLTL-BV encoding. This value is 1 if and only if

$$\begin{aligned} &(\exists i \in [l, l + 2^k - 1]. \pi, (i + \tau) \models \xi \text{ and} \\ &\quad \forall j \in [l, i - 1]. \pi, (j + \tau) \models \psi) \text{ or} \\ &(\forall x \in [l, l + 2^k - 1]. \pi, (x + \tau) \models \psi \text{ and} \\ &\quad (\exists y \in [l, l + 2^k - 1]. \pi, (y + \tau + 2^k) \models \xi \text{ and} \\ &\quad \quad \forall z \in [l, y - 1]. \pi, (z + \tau) \models \psi)) \\ \Leftrightarrow &(\text{Simplifying } y \text{ interval}) \\ &(\exists i \in [l, l + 2^k - 1]. \pi, (i + \tau) \models \xi \text{ and} \\ &\quad \forall j \in [l, i - 1]. \pi, (j + \tau) \models \psi) \text{ or} \\ &(\forall x \in [l, l + 2^k - 1]. \pi, (x + \tau) \models \psi \text{ and} \\ &\quad (\exists y \in [l + 2^k, l + 2^{k+1} - 1]. \pi, (y + \tau) \models \xi \text{ and} \\ &\quad \quad \forall z \in [l, y - 1]. \pi, (z + \tau) \models \psi)) \\ \Leftrightarrow &(\text{Combining } x \text{ and } z \text{ intervals}) \\ &(\exists i \in [l, l + 2^k - 1]. \pi, (i + \tau) \models \xi \text{ and} \\ &\quad \forall j \in [l, i - 1]. \pi, (j + \tau) \models \psi) \text{ or} \\ &(\exists y \in [l + 2^k, l + 2^{k+1} - 1]. \pi, (y + \tau) \models \xi \text{ and} \\ &\quad \forall z \in [l, y - 1]. \pi, (z + \tau) \models \psi) \\ \Leftrightarrow &(\text{Combining } i, y \text{ and } j, z \text{ intervals}) \\ &(\exists i \in [l, l + 2^{k+1} - 1]. \pi, (i + \tau) \models \xi \text{ and} \\ &\quad \forall j \in [l, i - 1]. \pi, (j + \tau) \models \psi) \\ \Leftrightarrow &\pi, \tau \models \psi \mathcal{U}_{[l, l+2^{k+1}-1]}\xi \end{aligned}$$

Proving the property for $k + 1$ and completing the proof. $\square$

POLYVER: A Compositional Approach for Polyglot System Modeling and Verification

Pei-Wei Chen^{*}, Shaokai Lin^{*}, Adwait Godbole^{*}, Ramneet Singh[†],
Elizabeth Polgreen[‡], Edward A. Lee^{*}, Sanjit A. Seshia^{*}

^{*}University of California, Berkeley, USA, {pwchen, shaokai, adwait, eal, sseshia}@berkeley.edu

[†]Indian Institute of Technology, Delhi, India, ramneet2001@gmail.com

[‡]University of Edinburgh, UK, elizabeth.polgreen@ed.ac.uk

Abstract—Many software systems are *polyglot*; that is, they comprise programs implemented in a combination of programming languages. Program verifiers, however, tend to be customized for individual languages. Verification by compiling to a common encoding requires supporting full language syntax and semantics which is prohibitive for modern languages. We present POLYVER, an alternative compositional approach to polyglot verification that bootstraps off-the-shelf language-specific verifiers with abstraction and synthesis.

POLYVER uses contracts written in an intermediate language to abstract individual procedures in the system. Our verification approach uses language-specific verifiers (e.g., for C or Rust) to validate these contracts and the UCLID5 model checker for compositionally verifying a temporal property on the overall system using the contracts. The intermediate language sidesteps the need for compiling implementation languages to a common encoding, a key obstacle with polyglot verification. Finally, POLYVER automates the generation of contracts using synthesis oracles such as large-language-models (LLMs). Overall POLYVER performs contract synthesis and verification in a counterexample-guided abstraction refinement and inductive synthesis (CEGIS-CEGAR) loop to verify the system-level property. We use POLYVER to verify programs in the Lingua Franca polyglot language. We are able to verify systems with C and Rust procedures, as well as C language fragments that were unsupported in previous work.

Index Terms—Polyglot languages, formal verification, synthesis, contracts, large language models.

I. INTRODUCTION

Many modern software systems comprise code written in a combination of programming languages [1]. For example, the Robot Operation System 2 (ROS2) [2], [3] allows users to specify processes that communicate with each other, called ROS nodes, where each node can be implemented in a different programming language. Other languages and frameworks provide similar cross-language functionality for various domains, like Lingua Franca [4] for real-time, distributed embedded systems, and Eclipse Cyclone DDS [5] and gRPC [6] for distributed systems and networking. We refer to these multi-language software systems as *polyglot* systems.

Polyglot systems allow developers to pick and choose the best programming language for specific system components. Unfortunately, this flexibility comes at a cost, with studies indicating that the increased complexity of polyglot systems can lead to an increase in bugs [1]. Recent work has addressed these issues through testing and other incomplete methods. For example, Yang et al. [7] develop a deep-learning-based method

that helps localize bugs specifically due to the interaction between multiple languages in a single system. These tools and techniques can help developers build safer systems, but they cannot provide the guarantees needed in safety-critical settings and other vital settings in which polyglot systems appear.

In this paper, we study the problem of formally verifying polyglot systems. Just like developing polyglot systems introduces new challenges in the development cycle, verifying polyglot systems introduces new challenges in the verification process. Over the past few decades, there has been substantial work on program verifiers that operate directly on individual programming languages. We refer to these tools as *language-specific verifiers*. For example, CBMC [8] and Frama-C [9] can be used to verify C code, Kani [10] and Verus [11] for Rust, ESC/Java [12], Java PathFinder [13], and JBMC for Java [14], etc. However, very few approaches have been proposed for multi-language systems, so polyglot verification challenges remain largely unexplored.

The main existing approach for polyglot system verification translates all code to one common language and applies existing reasoning engines to that common representation. However, this approach is challenging due to the complexity of handling diverse syntax and semantics across languages, and arguably, it loses the benefits of having highly optimized language-specific verifiers. For instance, prior work, such as Lin et al. [15], only handles very restricted C code within the Lingua Franca language, highlighting the difficulties of such monolithic approaches. Encoding a multi-language system in a single verification model may also result in scalability issues, as the generated verification problem can become prohibitively large. To our knowledge, no existing approach enables automated verification of multi-language software systems through *cooperative* use of language-specific verifiers.

We propose a compositional verification approach to address this gap. Our approach has two key components: (a) interface contracts written in an *intermediate contract language* that abstract individual components, and (b) a *synthesizer* that automatically generates and refines these contracts with the help of language-specific verifiers. The intermediate contract language is much smaller than the languages of individual components, allowing us to sidestep full language translation. Instead, we can use language-specific verifiers to check the contract validity by only translating the (much smaller) inter-

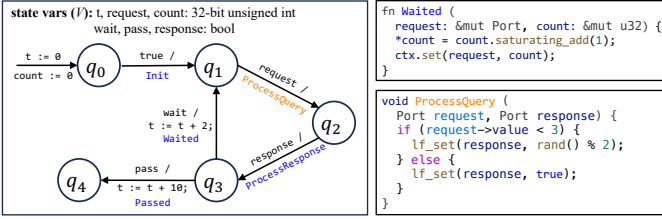


Fig. 1: An Extended State Machine (ESM) describing a subway traffic control system implemented in Lingua Franca (LF). a/b means that when a is present, b is run. The procedures in blue and orange are Rust and C procedures respectively. Implementations of two procedures are shown on the right. `ctx.set` and `lf_set` are LF API calls assigning values to ports.

mediate language fragment. Further, our large language model (LLM)-based synthesizer and language-specific verifiers eliminate the burden of manually writing and refining contracts. This enables scalable and automated verification of multi-language systems without requiring a full translation into a single verification framework.

In summary, the novel contributions of this paper are:

- **Polyglot verification problem.** We formalize the polyglot verification problem and propose a compositional solution integrating multiple language-specific verifiers.
- **Automated pre/post-condition contract synthesis.** We automate synthesis and refinement of pre/post-condition contracts using an intermediate language and combining counterexample-guided inductive synthesis (CEGIS) and abstraction refinement (CEGAR). Our approach uses synthesis oracles including syntax-guided synthesis (SyGuS) solvers and large language models (LLMs).
- **Our tool.** We implemented POLYVER on top of UCLID5 [16], [17], supporting C and Rust via CBMC [8] and Kani [10]. Our implementation shows that an LLM-based synthesis oracle outperforms pure symbolic synthesis approaches.
- **Case study and evaluation.** We developed a verifier for the Lingua Franca coordination language [4] and show that our approach is able to verify multi-language systems with reasonable synthesis time. We also introduce a new set of polyglot verification benchmarks and realistic applications that are only solved by our approach.

II. MOTIVATION AND BACKGROUND

Consider a train management system modeled in Figure 1 as an Extended State Machine (ESM) [18], i.e., a guarded transition system where transitions have procedures attached to them. Our example has procedures in two languages – the `Waited` procedure in Rust and `ProcessQuery` in C.

Encodings must match language semantics. Differences in language semantics can make encoding polyglot systems into a single formal model challenging. For example, the `x.saturating_add(1)` Rust function in `Waited` saturates addition overflows to the maximum integer value (for

```
fn Waited (
  request: &mut Port, count: &mut u32) {
  *count = count.saturating_add(1);
  ctx.set(request, count);
}

void ProcessQuery (
  Port request, Port response) {
  if (request->value < 3) {
    lf_set(response, rand() % 2);
  } else {
    lf_set(response, true);
  }
}
```

Fig. 2: UCLID5 translation of `x.saturating_add(1)` in Rust. Notice that the translation depends on the type of `x`. In UCLID5 and SMT-LIB, bitvectors do not have implicit type interpretation. Instead, they are only interpreted when specific operators are applied. For example, `< (bvslt)` and `<_u (bvult)` stands for the signed and unsigned less than comparators in UCLID (SMT-LIB), respectively.

that type) instead of wrapping around. Figure 2 shows the corresponding UCLID5 expression of this function. Encodings must faithfully capture *all language-specific constructs* which can be large/complex for modern languages.

Language-specific verification (LSV) tools such as CBMC [8] for C, Kani [19] for Rust, and Cameleer [20] for OCaml already perform this heavy lifting of faithfully capturing language semantics. However, these tools might use different backends/proof techniques. For example, CBMC and Kani perform symbolic bounded model checking, while Cameleer uses deductive verification. Verifying polyglot systems with LSVs requires *connecting these verifiers*, which is challenging due to these differences.

Compositional (contract-based) verification uses interface specifications, also called *contracts*, to abstract individual system components. Contracts can be assume-guarantee pairs for concurrent systems or pre/post-condition pairs for sequential software. Once the validity of a contract w.r.t. the system component is checked using an LSV, it can be soundly used for system-level verification. Crucially, given the contract, system-level check becomes *agnostic of techniques/backends employed by the LSV*. POLYVER uses this precise strategy of using contracts as the glue connecting language-specific verifiers. In this sense, verification in POLYVER operates much like assume-guarantee/requires-ensures reasoning [21], [22]. Manually identifying valid, yet *strong enough* contracts, however, is challenging. E.g., `ProcessQuery` (Fig. 1) requires a non-trivial postcondition:

```
((request->value < 3) || response->value)
  && (request == \old(request))
```

To address this, **POLYVER performs automated contract synthesis** by using refinement-based neural (LLM-based) contract search in combination with symbolic verification of the contract. Thus, with contract synthesis, POLYVER bootstraps existing LSVs to solve the multi-language verification problem, at the same time avoiding the challenges of encoding the entire system into a single language.

III. PROBLEM FORMULATION

Polyglot systems are compositions of components where each component is a procedure in some language $\mathcal{L}$ (e.g., C, Rust, Java). Such a setup is widely seen in distributed systems

where nodes execute code in a specific language and interact via mechanisms like remote procedure calls (RPCs). We formalize such systems and define the verification problem.

We model polyglot systems as Extended State Machines (ESM) [23]. Informally, ESMs capture inter-procedural control flow as transitions between *modes*, where each transition has an associated *update function* representing a sequence of procedure calls. ESMs operate on a set of typed variables V shared across all procedures in the system. We refer to the domain of (typed) variable $v \in V$ as $D(v)$ and denote the Cartesian product of the domains of all variables as $D(V)$. $D(V)$ thus forms the set of assignments d to V . A *predicate* is a function $D(V) \rightarrow \mathbb{B}$.

Definition 1 (Extended State Machine). *An Extended State Machine (ESM) is a tuple (Q, V, I_0, T) where*

- 1) Q is a set of modes,
- 2) V is a set of typed variables,
- 3) $I_0 \subseteq Q \times D(V)$ is a set of initial states,
- 4) $T \subseteq (Q \times Q) \times (G \times U)$ is a set of mode-mode transitions, each associated with a guard in $G = D(V) \rightarrow \mathbb{B}$ and an update relation in $U = 2^{D(V) \times D(V)}$.

Executions. The guard g is a predicate determining whether a transition can occur, and the update relation u defines how the transition affects V . At every step, the ESM in a state (q, d) takes a transition $((q, q'), (g, u))$ such that $g(d)$ is true. The ESM then updates the state to (q', d') where $(d, d') \in u$. While the update relation (u) in each ESM transition is kept abstract, polyglot systems specialize them to be invocations of procedures (possibly from different languages).

Definition 2 (Language). *A language $\mathcal{L}$ is a set of terms and predicates constructed through syntactic rules over a set of variables V . The language fixes the interpretation of each term $t \in \mathcal{L}$ and predicate $p \in \mathcal{L}$ denoted as $\llbracket t \rrbracket_{\mathcal{L}} : D(V) \rightarrow D(V)$ and $\llbracket p \rrbracket_{\mathcal{L}} : D(V) \rightarrow \mathbb{B}$ respectively.*

We assume that the set V contains all the variables read or written by the procedures implemented in the various languages. Further, the interface variables shared across languages must have matching types, and hence we use the same $D(V)$ for different languages. This interface type-consistency is enforced in practical polyglot frameworks such as Lingua Franca [4] and ROS2 [2]. Unless otherwise noted, we implicitly use the same variable set V for the rest of the paper.

Next, we define procedures.

Definition 3 (Procedure). *A procedure f is a pair $(C, \mathcal{L})$ where $C \in \mathcal{L}$ represents the procedure body. We call such a procedure an $\mathcal{L}$ -procedure. We refer to pre-state and post-state variables of a procedure as V and V' , respectively.*

We assume that procedures are total and always terminate. Polyglot models, our model for polyglot systems, are ESMs where update relations are procedure invocation sequences.

Definition 4 (Polyglot Model). *A polyglot model $\mathcal{M}$ is a tuple (Q, V, I_0, T, F) where (Q, V, I_0, T) is an ESM and F*

is a set of procedures. Each transition $((q, q'), (g, u)) \in T$ is s.t. $u = \{(d, d') \mid d' = f_k \circ \dots \circ f_1(d)\}$ for some $f_i \in F$, where $\circ$ denotes procedure composition: $f_1 \circ f_2(d) = \llbracket C_1 \rrbracket_{\mathcal{L}_1}(\llbracket C_2 \rrbracket_{\mathcal{L}_2}(d))$.

An example of a polyglot model is shown in Figure 1, where $Q = \{q_i\}_i$, $I_0 = (q_0, _)$ is the state after executing the procedures on the initial transition, T is depicted by the transitions, and $F = \{\text{Init}, \text{ProcessQuery}, \dots\}$.

Remark on semantics: While procedures are deterministic in our formulation (terms have deterministic semantics), non-deterministic procedures can be introduced by changing term interpretations from functions to relations. Further, side-effects are emulated by extending V with global variables.

Traces. Each execution of the model produces an infinite trace that is the sequence of assignments (d) to variables in V seen in the execution. A property ϕ is a set of traces, typically in the form of a temporal logic formula. Let the set of all traces that can be produced by a model $\mathcal{M}$ be $\mathcal{T}(\mathcal{M})$. We write $\mathcal{M} \models \phi$ if $\mathcal{T}(\mathcal{M}) \subseteq \phi$. The polyglot verification problem aims to determine whether a polyglot model satisfies a property ϕ .

Problem Statement 1 (Polyglot verification problem). *Given a polyglot model $\mathcal{M} = (Q, V, I_0, T, F)$ and a system property ϕ determine whether $\mathcal{M} \models \phi$. If $\mathcal{M} \not\models \phi$, provide a counterexample trace generated by $\mathcal{M}$ that violates ϕ .*

Verification Challenge and Insight. Crucially, while the procedure bodies (C) in $\mathcal{M}$ are known, capturing their interpretations ($\llbracket C \rrbracket_{\mathcal{L}}$) is difficult. Naive monolithic verification of $\mathcal{M}$ would require encoding full semantics of $\mathcal{L}$ (for all $\mathcal{L}$) to a common modeling language, e.g. SMT-LIB. This is extremely tedious and would require a parser/compiler infrastructure for each language. Our solution is to use contracts written in an intermediate language $\mathcal{L}^*$ – in this way we only need to support encodings for this (much smaller) language $\mathcal{L}^*$.

IV. POLYGLOT VERIFICATION THROUGH A CONTRACT IL

We now introduce contract-based decomposition. The key component is a pre-post condition contract that abstracts (over-approximates) the behavior of a procedure f .

Definition 5 (Contract). *A $(\mathcal{L})$ -contract for procedure $(C, \mathcal{L})$ is a pair (P, Q) where P, Q are predicates in $\mathcal{L}$. (P, Q) is valid iff the Hoare-triple $\{P\}C\{Q\}$ holds, i.e.,*

$$\forall d, d' \in D(V). (\llbracket P \rrbracket_{\mathcal{L}}(d) \wedge d' = \llbracket C \rrbracket_{\mathcal{L}}(d)) \implies \llbracket Q \rrbracket_{\mathcal{L}}(d') \quad (1)$$

By convention, preconditions P can only refer to pre-state variables V , while postconditions Q can refer to both pre- and post-state variables V and V' . Equation (1) can be checked with a language-specific verifier for $\mathcal{L}$. A valid contract can serve as a sound system-facing abstraction of the procedure, as we now discuss.

Contract composition replaces procedures in a polyglot system $\mathcal{M}$ with their contracts. Formally, for each transition

$u = f_k \circ \dots \circ f_1$ in $\mathcal{M}$, we generate contracts (P_j, Q_j) for each $\mathcal{L}_j$ -procedure f_j such that,

$$\forall d \in D(V). \llbracket Q_j \rrbracket_{\mathcal{L}_j}(d) \implies \llbracket P_{j+1} \rrbracket_{\mathcal{L}_{j+1}}(d) \quad (2)$$

Then, we define a new model $\mathcal{M}'$ where u is replaced with the contract-derived relation $u' = \{(a, b) \mid \llbracket P_1(a) \rrbracket_{\mathcal{L}_1} \implies \llbracket Q_k(b) \rrbracket_{\mathcal{L}_k}\}$. Since the replacement contract u' overapproximates u , we have the following theorem.

Theorem 1. *If $\mathcal{M}' \models \phi$, then $\mathcal{M} \models \phi$.*

How does a contract IL help verification? A key challenge with contract-based polyglot verification is checking Equation (2), which involves predicates in *two different languages*. We are unable to use a language-specific verifier to check this. Our solution is to *write contracts in an intermediate language $\mathcal{L}^*$* and bridge contracts using a *semantics-preserving compilation*.

Definition 6 (Intermediate Language). *An intermediate language (IL) $\mathcal{L}^*$ is a language of pre/post-condition contracts, whose semantics is defined by the SMT-based semantics of the language $\mathcal{L}$ to which it is translated.*

Concretely, we translate contract predicates from $\mathcal{L}^*$ to each procedure language $\mathcal{L}$, e.g., C, Rust, through a semantics-preserving compilation $\text{comp}_{\mathcal{L}}$:

Assumption 1 ($\mathcal{L}^* \rightarrow \mathcal{L}$ Compilation).

$$\forall C \in \mathcal{L}^*. \llbracket \text{comp}_{\mathcal{L}}(C) \rrbracket_{\mathcal{L}} \equiv \llbracket C \rrbracket_{\mathcal{L}^*}$$

Assumption 1 allows us to compose and validate contracts in $\mathcal{L}^*$. When contracts are in $\mathcal{L}^*$, contract composition check (Equation (2)) simplifies to:

$$\forall d \in D(V). \llbracket Q_j \rrbracket_{\mathcal{L}^*}(d) \implies \llbracket P_{j+1} \rrbracket_{\mathcal{L}^*}(d) \quad (3)$$

Equation (3) can be checked by a model checker when $\mathcal{L}^*$ is translated into a modeling language like UCLID5. Validity checking (Equation (1)) of an $\mathcal{L}^*$ -contract (P, Q) simply reduces to that of an $\mathcal{L}$ -contract $(\text{comp}_{\mathcal{L}}(P), \text{comp}_{\mathcal{L}}(Q))$.

Counterexample guided contract refinement. While Theorem 1 provides a soundness result, the converse does not hold. A counterexample trace found in $\mathcal{M}'$ may be a spurious counterexample that cannot be produced by $\mathcal{M}$. Contracts must be strengthened to exclude this trace. However, developing *valid contracts that are strong enough* to prove system-level properties is challenging, often involving iterative contract refinement. To address this, we introduce the valid contract synthesis problem, which aims to generate a contract that avoids a set of spurious traces provided as *negative examples*.

Problem Statement 2 (Valid Contract Synthesis Problem). *Given an $\mathcal{L}$ -procedure f and positive, negative example sets $X, \bar{X} \subseteq D(V) \times D(V)$, synthesize a valid contract in $\mathcal{L}^*$ that includes the positive examples and excludes negative ones.*

How does a contract IL help synthesis? Contract synthesis illustrates the second benefit of the intermediate language $\mathcal{L}^*$. Language-specific contracts would require synthesizers tailored to the syntax and semantics of each language. A

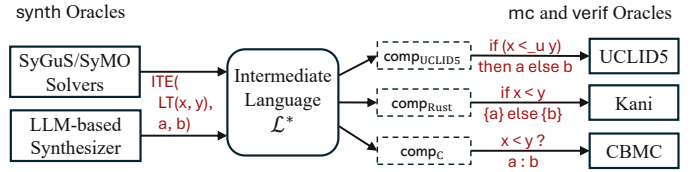


Fig. 3: An intermediate language $\mathcal{L}^*$ act as glue between languages. Synthesis oracles generate expressions in $\mathcal{L}^*$, which are translated to semantic equivalent expressions in specific languages used by a model checker or verification oracles. x and y are unsigned bitvectors, hence using $<_{\text{u}}$ in UCLID5.

common IL allows us to maintain a single interface with synthesis oracles. For instance, in the case of Syntax Guided Synthesis (SyGuS) [24] we require a single grammar, or in the case of neural search (e.g., using LLMs), we require a single in-context example set. Like verification, this greatly reduces the effort in developing synthesis oracles. Figure 3 illustrates how $\mathcal{L}^*$ mediates contract synthesis and verification.

V. AN ABSTRACTION SYNTHESIS APPROACH FOR POLYGLOT MODEL VERIFICATION

We present an approach for contract synthesis (Problem 2) which we use as a subroutine to solve Problem 1. We provide some background and then an overview in Section V-B.

A. Background: CEGIS and CEGAR

We synthesize contracts by combining Counterexample Guided Inductive Synthesis (CEGIS) [25] and Counterexample Guided Refinement Abstraction (CEGAR) [26].

CEGIS uses an iterative interaction between a *learner* and a *teacher* to synthesize a solution (in our case contracts) satisfying a set of constraints (in our case, example traces). The *learner* generates a candidate solution satisfying the constraints, while the *teacher* checks whether the solution satisfies the overall specification. Synthesis succeeds if the solution is correct; else, the teacher provides a new counterexample.

CEGAR iteratively refines an abstraction (in our case contracts) in the context of a property. At each step, the abstraction is checked for correctness. If verification fails, the counterexample is analyzed to determine whether it is spurious (due to the abstraction being too coarse). If the counterexample is real, the system is buggy; else, the abstraction is refined.

B. Approach Overview

We illustrate our CEGIS/CEGAR-hybrid approach to the polyglot verification in Figure 4 and Algorithm 1. The approach is parameterized by the set of synthesis oracles, verification oracles, and the intermediate contract language $\mathcal{L}^*$. It operates by performing CEGIS (inner loop; line 5) using synthesis and verification oracles to synthesize contracts for each procedure, which solves Problem 2. CEGAR (outer loop) then uses these contracts to generate the induced abstract model $\mathcal{M}'$ (line 6) specified entirely in $\mathcal{L}^*$ by combining the system model and the contracts. We then check these

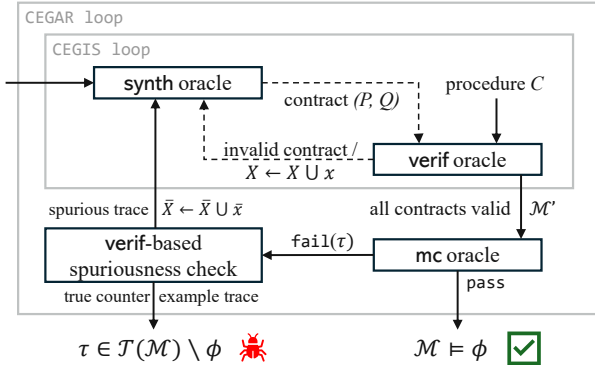


Fig. 4: Flow diagram of our approach. The process starts from the synthesis oracle. Contracts are synthesized in the inner (dashed) CEGIS loop and refined in the outer CEGAR loop. X and $\bar{X}$ denotes positive and negative example sets.

contracts against the system-level property (line 7) – if the property is proven, the system is verified (line 8); otherwise, we check if the counterexample trace is spurious (line 13). If the trace is spurious, we exclude it by returning to the inner CEGIS loop to refine contracts, else we report failure with the trace (line 20). Thus, POLYVER iteratively switches between CEGIS and CEGAR to generate a counterexample or prove the property.

Algorithm 1 PolyVer: CEGIS/CEGAR-hybrid approach

Require: $\mathcal{M} = (Q, V, I_0, T, F)$: a polyglot model, ϕ : a property

Ensure: res : verification result, τ : a counterexample trace

```

1: for all  $f_i \in F$  do
2:    $X_i, \bar{X}_i \leftarrow \emptyset, \emptyset$ 
3: end for
4: while  $\top$  do ▷ CEGAR loop
5:    $\mathcal{C} \leftarrow \{\text{synthContract}(f_i, X_i, \bar{X}_i)\}_{f_i}$  ▷ CEGIS loop
6:    $\mathcal{M}' \leftarrow \text{induce}(\mathcal{M}, \mathcal{C})$ 
7:    $res \leftarrow \text{mc}(\mathcal{M}', \phi)$ 
8:   if  $res = \text{pass}$  then ▷ Verification succeeds
9:     return  $\text{pass}, -$ 
10:  else if  $res = \text{fail}(\tau)$  then
11:     $isSpurious \leftarrow \perp$ 
12:    for all  $f_i \in F$  do
13:       $\bar{X}'_i \leftarrow \text{checkSpurious}(\tau, f_i)$ 
14:      if  $\bar{X}'_i \neq \emptyset$  then
15:         $isSpurious \leftarrow \top$ 
16:      end if
17:       $\bar{X}_i \leftarrow \bar{X}_i \cup \bar{X}'_i$ 
18:    end for
19:    if  $\neg isSpurious$  then ▷ Counterexample found
20:      return  $\text{fail}, \tau$ 
21:    end if
22:  end while
23: end while

```

C. Oracles

We assume access to synthesis and verification oracles, and a model checker. Synthesis oracles accept positive, negative example sets X , $\bar{X}$, and return an $\mathcal{L}^*$ -contract (P, Q) that satisfies X and dissatisfies $\bar{X}$:

$$\text{synth}(X, \bar{X}, C \in \mathcal{L}) \rightarrow (P \in \mathcal{L}^*, Q \in \mathcal{L}^*)$$

A verification oracle call $\text{verif}(P, Q, C)$ verifies $\mathcal{L}^*$ -contract validity w.r.t. an $\mathcal{L}$ -procedure (producing a *positive*-example if invalid): The verifier is a part of our trusted computing base (TCB) and is assumed to be sound and complete:

Assumption 2 (Verifier).

$$\text{verif}(P, Q, C) = \text{pass} \iff \{P\} C \{Q\}$$

$$\text{verif}(P, Q, C) = \text{fail}(x = (d, d')) \iff$$

$$\llbracket P \rrbracket_{\mathcal{L}^*}(d) \wedge d' = \llbracket C \rrbracket_{\mathcal{L}}(d) \wedge \neg \llbracket Q \rrbracket_{\mathcal{L}^*}(d')$$

A model checker call $\text{mc}(\mathcal{M}'[P_i, Q_i], \phi)$ verifies the abstract model $\mathcal{M}'$ (with contracts) w.r.t. property ϕ , and produces a (*negative*-) counterexample trace τ if violated. Examples Y_f can be extracted from τ for each procedure f :

Assumption 3 (Model Checker).

$$\text{mc}(\mathcal{M}'[P_i, Q_i], \phi) = \text{pass} \iff \mathcal{M}' \models \phi$$

$$\text{mc}(\mathcal{M}'[P_i, Q_i], \phi) = \text{fail}(\tau) \iff \tau \in \mathcal{T}(\mathcal{M}') \setminus \phi$$

$$\text{with } \text{extract}(\tau) = \{Y_f\}_f$$

We now detail contract synthesis and refinement in Sections V-D and V-E respectively. Finally, we discuss correctness guarantees in Section V-F.

D. Contract Synthesis via CEGIS Loop

The (inner) CEGIS loop iterates to generate a valid contract for procedure $(C, \mathcal{L}) \in F$, described in Algorithm 2. In each iteration, given $C \in \mathcal{L}$, and initially empty example sets X and $\bar{X}$, $\text{synth}(X, \bar{X}, C)$ generates a contract (P, Q) in $\mathcal{L}^*$. Then, verif checks contract validity with respect to the procedure C . If $\text{verif}(P, Q, C) = \text{pass}$ the contract is valid; otherwise the positive example x is added to X and the loop continues until a valid contract is found. Under Assumption 2, this produces valid contracts upon termination, thereby solving Problem 2.

Example 1 (Contract synthesis). The below table illustrates CEGIS-based contract synthesis for `ProcessQuery`. `synth` synthesizes (P_1, Q_1) which `verif` checks for validity returns a positive example (d, d') . Note that the pre-state d satisfies P_1 but the post-state d' violates Q_2 . `synth` then refines the contract based on (d, d') and generates (P_2, Q_2) , which is again checked by `verif`. `req` and `resp` are short for request and response respectively.

Oracle	Output
synth iter. 1	$P_1 : \top, Q_1 : (\text{resp} == (\text{req} \geq 3))$
verif iter. 1	$\text{fail} / (\text{req}, \text{resp}) : d = (1, \perp), d' = (1, \top)$
synth iter. 2	$P_2 : \top, Q_2 : (\text{req} < 3 \mid \mid \text{resp} == \top)$
verif iter. 2	$\text{pass} / -$

Algorithm 2 $\text{synthContract}(f, X, \bar{X})$

Require: $f = (C, \mathcal{L})$: a procedure**Modify:** $X, \bar{X}$: a set of positive/negative examples**Ensure:** (P, Q) : a valid contract for f

```
1: while  $\top$  do
2:    $(P, Q) \leftarrow \text{synth}(X, \bar{X}, C)$ 
3:    $\text{res} \leftarrow \text{verif}(P, Q, C)$ 
4:   if  $\text{res} = \text{pass}$  then  $\triangleright$  Contract is valid
5:     return  $(P, Q)$ 
6:   else if  $\text{res} = \text{fail}(x)$  then  $\triangleright$  Contract is invalid
7:      $X \leftarrow X \cup \{x\}$   $\triangleright$  Collect positive example
8:   end if
9: end while
```

Valid contracts are then used to construct $\mathcal{M}'$ (§IV) which is then checked by mc. This may either succeed or return a counterexample trace, which is then checked for spuriousness.

E. Spuriousness Check

A counterexample trace generated by mc may be *spurious*, i.e., unreproducible by the procedures. We illustrate the checking process in Algorithm 3. We first extract pre/post variable assignments $p = \{(d, d')\}_i$ for each procedure call in the trace (line 2). For each $(d, d') \in p$, we check whether $\{V = d\} C \{V' \neq d'\}$ holds using *verif*. If it holds, the pair cannot be produced by the procedure and is then added as a negative example to $\bar{X}$ (“spurious trace” edge in Figure 4; line 5). This will be excluded during the next CEGIS loop refinement iteration. If all pre-post transitions are valid for all procedures, the trace is a true counterexample (“true cex trace” edge). Example 2 shows an example of a spuriousness check performed on *ProcessQuery* against a pair of assignments.

Algorithm 3 $\text{checkSpurious}(\tau, f)$

Require: τ : a counterexample trace, $f = (C, \mathcal{L})$: a procedure**Ensure:** $\bar{X}$: a set of negative examples for f

```
1:  $\bar{X} \leftarrow \emptyset$ 
2: for all  $(d, d') \in \text{extract}(\tau)$  do
3:    $\text{res} \leftarrow \text{verif}(V = d, V' \neq d', C)$ 
4:   if  $\text{res} = \text{pass}$  then  $\triangleright \tau$  is spurious
5:      $\bar{X} \leftarrow \bar{X} \cup \{(d, d')\}$   $\triangleright$  Collect negative example
6:   end if
7: end for
8: return  $\bar{X}$ 
```

Example 2 (Spuriousness check). Continuing from Example 1, assume that a model checker (mc) checks the property using (P_2, Q_2) and returns a trace containing d and d' . To check spuriousness, we check the Hoare-triple $\{(\text{req}, \text{resp}) = d\} \text{ProcessQuery} \{(\text{req}', \text{resp}') \neq d'\}$ using *verif*, who returns *pass*. The pair is thus impossible, making the trace spurious. We then return to the CEGIS loop and re-synthesize a contract, which is proven valid by *verif*.

Oracle	Output
mc in CEGAR	$(\dots, d = (2, \top), d' = (10, \top), \dots)$
verif in CEGAR	$\text{pass} / (\text{req}, \text{resp}) : d = (2, \top), d' = (2, \top)$
synth in CEGIS	$P_3 : \top, Q_3 : (\text{req} < 3 \mid \mid \text{resp} == \top) \wedge \text{old}(\text{req}) == \text{req}$
verif in CEGIS	$\text{pass} / -$

F. Correctness Guarantee

Our approach makes assumptions 1, 2, and 3. Given these assumptions, we guarantee *partial correctness*, meaning that if POLYVER terminates with *pass* result, the system is verified, $\mathcal{M} \models \phi$ (soundness), and if it terminates with a *fail* result, the system is unsafe, $\mathcal{M} \not\models \phi$ (completeness).

We intuit this result as follows. For soundness, notice that contracts used in the induced model must overapproximate procedure behaviors because of the verification oracle assumption (Assumption 2). This means that when mc reports *pass*, there does not exist a trace that is producible by the procedures that violates the property, since all producible behaviors have already been captured by the contracts. For completeness, when mc reports *fail* with a trace, the trace must violate the property. This trace must also be producible by the procedures by the spuriousness check (§V-E).

VI. POLYVER IMPLEMENTATION

We implement the POLYVER automated approach in a tool¹ which builds on top of the UCLID5 verifier [16] and currently supports the C and Rust languages. The tool is modular and can be extended to support other languages and verifiers. Users need to provide oracles that subscribe to certain interfaces, which we now describe. We also illustrate oracles we have used for C and Rust. We use Boolean connectives, arithmetic operations, and comparators for the intermediate language $\mathcal{L}^*$.

A. Synthesis Oracle Interface

Given a procedure C , a set of positive and negative examples $X, \bar{X}$, the synthesis oracle *synth* returns an $\mathcal{L}^*$ -contract that is consistent with the examples. Below we discuss two candidate synthesis oracles and their interfaces.

1) *LLM-based Synthesizer*: LLMs are a good fit for contract synthesis because of their understanding and reasoning abilities across multiple (common) programming languages. We developed an LLM-based synthesizer that uses chain-of-thought prompting [27] with error and example feedback to generate contracts, i.e., the LLM is asked to first summarize the behavior of C and then generate a contract in $\mathcal{L}^*$.

Inspired by work by Mora et al. [28], we formulate a Python-based DSL that allows the LLM to construct $\mathcal{L}^*$ -expressions in a familiar language. We note that this does not affect the expressivity of generated the contract expressions. However, it increases the LLM’s ability to generate syntactically correct expressions. If the expression is invalid, we re-prompt the LLM with an error message indicating incorrect syntax or inconsistent types. Provided with positive and negative examples, the LLM is asked to find a constraint

¹Available at <https://github.com/uclid-org/polyver>.

that distinguishes the two sets. Details of our prompt are provided in Appendix A.

2) *SyGuS/SyMO-based Synthesizer*: The second oracle treats the problem as a Programming by Example (PBE) problem [29], asserting that the generated contract (P, Q) must² satisfy examples in X and dissatisfy those in $\bar{X}$:

$$\begin{aligned} \forall (d, d') \in X. \llbracket P \rrbracket_{\mathcal{L}^*}(d) \implies \llbracket Q \rrbracket_{\mathcal{L}^*}(d') \\ \wedge \forall (d, d') \in \bar{X}. \llbracket P \rrbracket_{\mathcal{L}^*}(d) \wedge \neg \llbracket Q \rrbracket_{\mathcal{L}^*}(d') \end{aligned}$$

We use Syntax-Guided Synthesis (SyGuS) [24] solvers to solve the PBE problems. A similar approach is to encode as a SyGuS Modulo Oracles (SyMO) [30] problem, a variant of SyGuS that allows constraints to contain black-box functions. We use SyMO solvers with language-specific verifiers as black-box functions. As the solver itself forms a CEGIS loop, we can directly solve Problem 2 in a single iteration.

B. Verification Oracle Interface

We use CBMC and Kani as verification oracles for C and Rust, respectively. Given a procedure body C and a contract (P, Q) , we generate CBMC and Kani queries by constructing $\{P\}C\{Q\}$ using their respective APIs. Listing 1 shows an example of a CBMC query. Specifically, at lines 20 and 22, we use CBMC APIs to assume and assert the PRECONDITION and POSTCONDITION C macros around the function call at line 21. This query is then verified using CBMC. We apply a similar template for Rust procedures, verified with the Kani model checker. CBMC and Kani can achieve unbounded correctness for programs with loops by using loop invariants or loop unwinding assertions.

```

1 // ... Definitions of structs and APIs ...
2 void ProcessQuery() {
3     if (query->value < 3) {
4         lf_set(response, rand() % 2 == 1);
5     } else { lf_set(response, true); }
6 }
7 int main() {
8     // alloc for inputs and the self struct.
9     query = calloc(1, sizeof(UMS_query_t));
10    response = calloc(1, sizeof(UMS_response_t));
11    // Assume that there are no NULL pointers.
12    __CPROVER_assume(query && response);
13    // Initialize with nondeterministic values.
14    *query = nondet_UMS_query_t();
15    // CBMC checks pre/post-conditions.
16    __CPROVER_assume(PRECONDITION);
17    ProcessQuery();
18    assert(POSTCONDITION);
19 }

```

Listing 1: CBMC query generated for ProcessQuery

VII. CASE STUDY: LF PROGRAM VERIFICATION

We demonstrate POLYVER’s effectiveness by developing a verifier for the Lingua Franca (LF) coordination language [4]. LF is a polyglot coordination language designed for real-time,

concurrent, and distributed systems. LF programs contain concurrent components called *reactors* connected through *ports* and *connections*. A reactor contains reactive message handlers called *reactions*, which can be written in a target programming language, e.g., C, C++, Rust, Python, or TypeScript. A reaction is invoked when any of its triggers are present, and it may produce effects triggering downstream reactions. With multiple reaction languages, Lingua Franca verification exhibits all the polyglot system verification challenges discussed previously.

A. Our Running Example as a Lingua Franca Program

Figure 5 shows the full LF program for our running example (Fig. 1), a subway U-turn management system (UMS). The *Train* and *UMS* reactors, written in Rust and C respectively, use special brackets $\{ \dots \}$ to enclose reaction bodies. The system describes a train asking for permission to cross. The train sends out requests to the UMS, which decides whether to grant permission (*ProcessQuery*). If the train has previously made more than three attempts, the UMS will certainly grant permission; otherwise permission is granted randomly. If granted, the train will pass, taking ten minutes; otherwise, it waits two minutes before retrying (*ProcessResponse*). The property (ϕ) that the train passes in 12 minutes is false, since the train may take up to 16 minutes.

B. Methodology: Mapping LF Programs to Polyglot Models

LFVerifier [15] performs end-to-end LF verification by mapping the program onto a State Space Diagram (SSD) and compiling it along with the reaction body (C code) into a monolithic axiomatic model. POLYVER instead compiles the SSD into an operational polyglot model.

An SSD is a graph where each node (state) s contains the current logical time, a set R of reactions to be executed, and a set of pending events, and each edge represents logical time advancement. Figure 6 in the Appendix shows an example SSD generated from the LF program in Figure 5.

LF timing semantics at each state s impose a directed-acyclic graph (DAG) over the set of reactions R , with edges encoding execution precedence. A reaction may fire only when all of its predecessors have run to completion, so any topological sort of the DAG is a valid execution order. Independent reactions whose predecessors have finished may execute in either order, hence introducing nondeterministic reaction interleaving. Once all reactions in R are completed, the SSD state can advance to the next state.

We encode time progression and nondeterministic reaction interleaving in UCLID5 by encoding reaction DAGs and dynamically tracking their execution status. This yields a nondeterministic polyglot model (Definition 4) where each transition executes one reaction, as in the model in Figure 1. We integrate the POLYVER-based verifier into the LF compiler, which generates C, Rust, and UCLID5 code and performs verification.

²Note that unlike Assm. 2, this is not a part of the trusted base of the POLYVER approach. In particular, the LLM-based synthesizer (Sec. VI-A1) may not satisfy this.

```

1 // Rust reactor
2 reactor Train {
3   input response : bool
4   output request : u32
5   state count : u32 = 0
6   state passed : bool = false
7   logical action wait(2 minute)
8   logical action pass(10 minute)
9
10  reaction Init (startup) -> request {=
11    ctx.set(request, self.count);
12  =}
13  reaction Waited (wait) -> request {=
14    self.count = self.count.saturating_add(1);
15    ctx.set(request, self.count);
16  =}
17  reaction Passed (pass) {= self.passed = true; =}
18  reaction ProcessResponse (response) -> pass, wait {=
19    if ctx.get(response).unwrap() {
20      ctx.schedule(pass, 0);
21    } else {
22      ctx.schedule(wait, 0);
23    }
24  =}
25 }

```

```

26 // C reactor
27 reactor UMS {
28   input request : uint32_t
29   output response : bool
30
31   reaction ProcessQuery (request) -> response {=
32     if (request->value < 3) {
33       lf_set(response, rand() % 2);
34     } else { lf_set(response, true); }
35   =}
36 }
37
38 // Property to be verified
39 @property(
40   name="pass_within_12_seconds",
41   spec="F[0, 12 minute](t.passed)",
42 )
43 main reactor {
44   // Component instantiations
45   u = new UMS()
46   t = new Train()
47   // Component interactions
48   t.request -> u.request
49   u.response -> t.response
50 }

```

Fig. 5: LF program for a subway control system from [31].

VIII. EXPERIMENTAL EVALUATION

Our experimentation evaluates POLYVER on both its components: contract synthesis and verification using synthesized contracts. We want to answer the following research questions: **RQ1**. How does POLYVER compare to prior LF verification work? **RQ2**. Can POLYVER handle full-fledged LF examples? **RQ3**. Can POLYVER verify multi-language systems?

Experiment Setup. We run all experiments on a single core on a 3.7GHz Intel Core i9 machine with 62GB RAM. We generate contracts using an LLM-based synthesizer (OpenAI o1-mini) with three parallel LLM queries per procedure, and use CBMC-6.3.1, Kani-0.56.0, and z3-4.8.7 for verification. Parallel synthesis for multiple procedures is implemented in POLYVER but not used in the experiments. Since LF reactions often include unmodified variables irrelevant to strong/tight contracts, our LLM prompt includes a single instruction emphasizing the importance of identifying these variables.

A. RQ1: Comparison to prior LF verification work

We contrast POLYVER with LFVerifier [15] using its proposed LF benchmarks.³ These are arguably toy examples due to the limited C syntax [15] supports – only assignments, conditionals (incomplete), and arithmetic and Boolean operations. They use a subset of safety MTL properties [32]. A direct comparison is not entirely fair since [15] uses a monolithic, axiomatic encoding specialized for LF programs, while POLYVER uses a compositional, operational encoding that is more general. Despite this, [15] is, to our knowledge, the only prior work that performs end-to-end automated LF program verification, so we compare with it.

We report on benchmark details, synthesis iteration counts, and runtimes in Table I. POLYVER successfully synthesizes

TABLE I: The synthesis oracle time (SOT) is the total contract synthesis time for all procedures (proc) across all CEGIS iterations (IS). The verification oracle time (VOT) is the total time on verification oracles, including contract validation and spuriousness checks. The verification time (UT) is the total time on UCLID5, called once per CEGAR iteration (AR).

Benchmark	#proc	LOC		#Iters		CPU Time (s)			
		C	LF	IS	AR	SOT	VOT	UT	[15]
ADASModel	7	16	73	7	1	220.9	3.2	21.5	24.1
AircraftDoor	3	10	38	3	1	82.1	1.2	2.0	6.1
Alarm	2	12	30	2	1	62.3	0.9	2.1	5.8
CoopSchedule	2	2	37	2	1	54.5	0.8	13.4	15.5
Election	3	7	35	3	1	124.2	1.4	83.9	35.7
Election2	2	6	31	3	2	142.8	1.5	11.7	11.4
Elevator	13	53	206	13	1	413.1	33.4	664.8	106.1
Factorial	3	11	36	3	1	138.2	1.9	63.0	42.4
Fibonacci	5	14	39	5	1	167.3	2.4	390.9	264.3
PingPong	4	8	62	4	1	131.0	1.8	8.2	12.0
Pipe	5	10	60	5	1	147.4	2.4	172.2	144.1
ProcessMsg	4	10	32	4	1	116.6	1.8	5.1	14.6
ProcessSync	1	2	18	1	1	35.2	0.4	1.7	5.0
Railroad	10	64	66	17	2	678.9	9.0	264.8	223.4
Ring	4	7	45	4	1	105.6	1.8	6.8	16.7
RoadsideUnit	8	26	81	8	1	259.0	3.8	25.3	44.0
SafeSend	4	12	53	4	1	124.3	1.8	2.3	6.9
Subway	10	20	64	10	1	266.5	4.5	67.2	35.0
Thermostat	6	29	45	6	1	192.6	2.6	3.5	12.8
TrafficLight	4	46	42	8	2	462.1	4.4	216.1	-
TrainDoor	3	7	50	3	1	74.5	1.2	2.1	5.8
UnsafeSend	4	12	53	4	1	122.1	1.8	2.9	6.8

the contracts and verifies *all* benchmarks automatically, while [15] is unable to verify the TrafficLight example. In 18 out of the 22 benchmarks, the contracts were synthesized correctly in a single CEGIS loop, taking 37 seconds on average. For the rest, we observe that all but one reaction still requires one CEGIS iteration while one (hard) reaction requires all the remainder of iterations. The hard reaction often includes complex control flow and data dependencies, which

³While these benchmarks may be present in the LLM training data, no contracts are available.

makes it difficult even for human experts to reason about.

We record the synthesis oracle time (SOT), verification oracle time (VOT), and verification runtime (UT). Though the total runtime (SOT + VOT + UT), dominated by the contract synthesis time, is generally slower than that of [15], we argue that this is time well spent as POLYVER offers a significant **qualitative improvement** over [15]: There is no longer a need to develop a translator from a target language to the modeling language (e.g., C to UCLID5). Since we leverage robust language-specific verifiers (e.g., CBMC, Kani), we are also able to support a much more comprehensive fragment of C/Rust. For example, CBMC officially supports ANSI-C, while [15] develops a custom C-to-UCLID5 translator for a small C subset (which is what these benchmarks use). These benefits are highlighted in Section VIII-B and VIII-C.

We also experimented with SyGuS/SyMO solvers but found that these approaches required a large number of CEGAR iterations and failed to terminate in most benchmarks. We attribute this to the solvers’ lack of access to procedure bodies, relying solely on positive and negative examples for guidance.

B. RQ2: Can POLYVER handle full-fledged LF examples?

We now evaluate POLYVER on examples requiring a broader fragment of C that [15] is unable to verify.

We evaluate our approach on two embedded systems – a blinking LED controller and a hill-climbing robot – and a satellite altitude controller [33], [34]. The LED controller tracks blinking status, while the robot checks if it is in climbing mode when on a ramp. Both are resource-constrained embedded systems. The satellite example uses a PID controller and disables motors if user input exceeds safe limits. When library functions (e.g., `atan`) are unavailable, we overapproximate them using nondeterministic input/output values.

The results in Table II show that our approach is able to handle language complexities and generate contracts for actual implementation code. In the LED and hill climbing examples, the LLM-based synthesizer was able to summarize procedure behavior while ignoring details about low-level code when they are irrelevant. In the satellite example, the LLM struggled to synthesize a valid contract for the PID controller. However, we noticed that providing a simple comment in the code that captures the logic of the controller greatly helped synthesis. The synthesized contracts can all be checked by CBMC. This highlights POLYVER’s potential to verify real-world systems.

C. RQ3: Can POLYVER verify multi-language systems?

We show that POLYVER is able to verify multi-language LF programs, ROS2 applications, and programs with inter-language function calls via a foreign function interface (FFI). Note that these benchmarks cannot be solved by [15] as they involve multiple languages. Some results are shown in Table II.

The final set of LF benchmarks consists of the 22 examples in Section VIII-A with half of the C reactions replaced by Rust reactions, plus `Submit` and `TrainPass` (the running example in Figure 5). As the performance is similar for the 22 examples, we only report results for the two new benchmarks

TABLE II: Results for full-fledged and multi-language examples. The `Blink`, `HillClimb`, and `Satellite` examples include real implementation C code. The rest and other 22 variants in Table I are multi-language systems only solvable by POLYVER.

Benchmark	#proc	LOC		#Iters		CPU Time (s)		
		C/Rust	LF	CEGIS	CEGAR	SOT	VOT	UT
<code>Blink</code>	1	123	35	1	1	31.0	0.45	1.26
<code>HillClimb</code>	12	75	102	20	1	685.4	74.9	9.7
<code>Satellite</code>	6	424	166	9	2	729.0	32.2	7.0
<code>Submit</code>	7	43	66	7	1	276.7	4.8	9.4
<code>TrainPass</code>	5	14	44	5	1	160	8.8	14.1
<code>Service</code>	4	8	-	4	1	118.9	7.1	3.5
<code>FFI</code>	3	30	-	3	1	73.9	6.5	1.75

in Table II. The LLM-based synthesizer is able to reason about Rust code and library functions in both C and Rust.

Aside from LF, we can also model and verify ROS2 applications using services (the call-and-response model), shown by the `Service` example. The publish-subscribe pattern can be modeled as well. Another general class of polyglot system involves inter-language function calls, e.g., Rust using C via FFI or Java via Java Native Interface (JNI). These systems can be modeled by splitting programs into language-specific segments with typing interfaces (FFI example in Table II).

IX. RELATED WORK

Most approaches to verifying multi-language systems rely on manually translating some parts of the system until everything is in a common language. For instance, Cook et al. [35] translate fragments of assembly code into C to verify the Xen hypervisor, and Marin et al. [36] define a small imperative language within the Maude modeling language to manually encode LF reactions. Where it is common to combine a fragment of one language with another, tools may automate translation of that fragment. For example, prior work [15] on LF verification automatically translates a fragment of C into the verification language, and tools like CBMC [8] support small fragments of assembly code. Another method is to black-box unanalyzable code and infer summaries from their observed input-output behavior [37]. While similar to us, this relies on sufficient examples and lacks the guarantees that fully integrating language-specific verifiers provides.

We employ compositional verification based on Hoare logic [38]. This is commonly applied to verification problems that, whilst written in only one language, are too large to be tackled in a single verification query [39]. Previous work, e.g., [40]–[43], automates this by learning pre/post-conditions using abstract interpretation and CEGAR [26].

There are also property-guided approaches to inferring contracts, while we aim to discover a contract that is just good enough to prove a given property. Here, previous work uses syntactic patterns [44], Constrained Horn Clause solvers [45], and even CEGIS [25], [46]. Recently, LLMs have been applied to learning specifications, including pre/post-conditions, for code [47]. The work closest to ours uses CEGIS and LLMs to infer post-conditions, but neither of these approaches addresses the challenge of multi-language verification.

Our approach to compositional verification, relying on learning-based synthesis oracles, implements the paradigm of “verification by reduction to synthesis” and the use of learning-driven synthesis advocated by [48]. Our contract learning approach is based on an integration of CEGIS and CEGAR. This integration can be considered a form of oracle-guided synthesis/learning [30], [48], [49], where the verification oracle and model checker act as oracles guiding the synthesis search. In hardware, prior work [50] uses synthesis to generate environment invariants for modular hardware verification. These invariants can be seen as pre-conditions in our context.

X. CONCLUSIONS

We formulated the polyglot verification problem and proposed POLYVER, a contract-based approach. This approach bootstraps off-the-shelf language-specific verifiers to verify polyglot systems, while side-stepping challenges involved in compiling the entire system to a single homogenous encoding. We implemented our approach in a tool that automatically iterates between CEGAR and CEGIS loops to synthesize contracts using an LLM-based synthesizer. We demonstrate a significant qualitative improvement by verifying full-fledged, multi-language systems that previous work could not handle.

There are several interesting directions for future work. One is to apply the POLYVER approach to verifying other kinds of polyglot distributed software systems or robotics software systems. We believe our approach is also applicable in cases where the software is not necessarily polyglot to begin with. For example, it would also be interesting to apply this approach to the problem of formally-verified translation of code from one language to another (e.g., from C to Rust), where the translation is performed in stages, and the intermediate state of the translated code involves a mix of languages.

ACKNOWLEDGEMENTS

This work was supported in part by the DARPA Provably Correct Design of Adaptive Hybrid Neuro-Symbolic Cyber Physical Systems (ANSR) program award number FA8750-23-C-0080; the National Science Foundation (NSF) award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems); Nissan and Toyota under the iCyPhy Center, Intel, Berkeley Deep Drive, and the RDI Frontier Fellows Program.

REFERENCES

- [1] H. Yang, Y. Nong, S. Wang, and H. Cai, “Multi-language software development: Issues, challenges, and solutions,” *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 512–533, 2024.
- [2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, “ROS: An open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.
- [3] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>
- [4] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, “Toward a lingua franca for deterministic concurrent systems,” *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 4, May 2021. [Online]. Available: <https://doi.org/10.1145/3448128>
- [5] Eclipse Cyclone DDS project. [Online]. Available: <https://github.com/eclipse-cyclonedds/cyclonedds>
- [6] K. Indrasiri and D. Kuruppu, *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O’Reilly Media, 2020.
- [7] H. Yang, Y. Nong, T. Zhang, X. Luo, and H. Cai, “Learning to detect and localize multilingual bugs,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2190–2213, 2024.
- [8] D. Kroening, P. Schrammel, and M. Tautschnig, “CBMC: the C bounded model checker,” *CoRR*, vol. abs/2302.02384, 2023.
- [9] P. Cuoc, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-c: A software analysis perspective,” in *International conference on software engineering and formal methods*. Springer, 2012, pp. 233–247.
- [10] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, “Verifying dynamic trait objects in rust,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 321–330. [Online]. Available: <https://doi.org/10.1145/3510457.3513031>
- [11] 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,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.
- [12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended static checking for java,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)*, 2002, pp. 234–245.
- [13] K. Havelund and T. Pressburger, “Model checking java programs using java pathfinder,” *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 366–381, 2000.
- [14] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, “Jbmc: A bounded model checking tool for verifying java bytecode,” in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 183–190.
- [15] S. Lin, Y. A. Manerkar, M. Lohstroh, E. Polgreen, S.-J. Yu, C. Jerad, E. A. Lee, and S. A. Seshia, “Towards building verifiable cps using lingua franca,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 5s, Sep. 2023. [Online]. Available: <https://doi.org/10.1145/3609134>
- [16] E. Polgreen, K. Cheang, P. Gaddamadugu, A. Godbole, K. Laeuffer, S. Lin, Y. A. Manerkar, F. Mora, and S. A. Seshia, “UCLID5: multi-modal formal modeling, verification, and synthesis,” in *34th International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 13371. Springer, 2022, pp. 538–551.
- [17] S. A. Seshia and P. Subramanyan, “Uclid5: Integrating modeling, verification, synthesis, and learning,” in *Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, October 2018.
- [18] V. S. Alagar and K. Periyasamy, *Extended Finite State Machine*. London: Springer London, 2011, pp. 105–128. [Online]. Available: https://doi.org/10.1007/978-0-85729-277-3_7
- [19] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, “Verifying dynamic trait objects in rust,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 321–330. [Online]. Available: <https://doi.org/10.1145/3510457.3513031>
- [20] M. Pereira and A. Ravara, “Cameleer: A deductive verification tool for ocaml,” in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 677–689.
- [21] C. B. Jones, “Tentative steps toward a development method for interfering programs,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 4, pp. 596–619, 1983.
- [22] B. Liskov, J. Guttag *et al.*, *Abstraction and specification in program development*. MIT press Cambridge, 1986, vol. 180.
- [23] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. MIT Press, 2016. [Online]. Available: <http://leeseshia.org>
- [24] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, “Syntax-guided synthesis,” in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 1–8.
- [25] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, “Combinatorial sketching for finite programs,” in *ASPLOS*. ACM, 2006, pp. 404–415.

- [26] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, vol. 1855. Springer, 2000, pp. 154–169.
- [27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [28] F. Mora, J. Wong, H. Lepe, S. Bhatia, K. Elmaaroufi, G. Varghese, J. E. Gonzalez, E. Polgreen, and S. A. Seshia, "Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024. [Online]. Available: <https://openreview.net/forum?id=kQPzFwVlu>
- [29] S. Gulwani, "Programming by examples: Applications, algorithms, and ambiguity resolution," in *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8*. Springer, 2016, pp. 9–14.
- [30] E. Polgreen, A. Reynolds, and S. A. Seshia, "Satisfiability and synthesis modulo oracles," in *VMCAI*, vol. 13182. Springer, 2022, pp. 263–284.
- [31] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow language lustre," *IEEE transactions on software engineering*, vol. 18, no. 9, pp. 785–793, 1992.
- [32] J. Ouaknine and J. Worrell, "Safety metric temporal logic is fully decidable," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns and J. Palsberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 411–425.
- [33] J. Cardoso, C. Chanel, P. Chauvin, A. Hostallier, J. Lamaison, A. Mascarenas-Gonzales, E. Metral, and L. Alloza, "Lab on the real-time control of reaction wheel," 2022, unpublished lab exercise for the course IMAE803: Real-time Control of Aerospace Systems, M.Sc in Aerospace Engineering, ISAE-SUPAERO.
- [34] S. Lin, E. Jellum, M. Theile, T. Tanneberger, B. Sun, C. Jerad, R. Xu, G. Feng, C. Menard, M. Lohstroh, J. Castrillon, S. Seshia, and E. Lee, "Pretvm: Predictable, efficient virtual machine for real-time concurrency," 2024. [Online]. Available: <https://arxiv.org/abs/2406.06253>
- [35] B. Cook, B. Döbel, D. Kroening, N. Manthey, M. Pohlack, E. Polgreen, M. Tautschnig, and P. Wiecekiewicz, "Using model checking tools to triage the severity of security bugs in the xen hypervisor," in *FMCAD*. IEEE, 2020, pp. 185–193.
- [36] M. Marin, P. C. Ölveczky, M. Reja, M. Rukhaia, and K. Bae, *Semantics and Formal Analysis of Lingua Franca CPS Specifications in Rewriting Logic*. Cham: Springer Nature Switzerland, 2025, pp. 70–101. [Online]. Available: https://doi.org/10.1007/978-3-031-85134-6_4
- [37] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *ESOP*, vol. 9032. Springer, 2015, pp. 158–182.
- [38] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [39] D. Beyer, M. Kettl, and T. Lemberger, "Decomposing software verification using distributed summary synthesis," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1307–1329, 2024.
- [40] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo, "Automatic inference of necessary preconditions," in *VMCAI*, vol. 7737. Springer, 2013, pp. 128–148.
- [41] Y. Moy, "Sufficient preconditions for modular assertion checking," in *VMCAI*, vol. 4905. Springer, 2008, pp. 188–202.
- [42] M. Gordon and H. Collavizza, "Forward with hoare," in *Reflections on the Work of C. A. R. Hoare*. Springer, 2010, pp. 101–121.
- [43] M. N. Seghir and D. Kroening, "Counterexample-guided precondition inference," in *ESOP*, vol. 7792. Springer, 2013, pp. 451–471.
- [44] E. Denney and B. Fischer, "A generic annotation inference algorithm for the safety certification of automatically generated code," in *GPCE*. ACM, 2006, pp. 121–130.
- [45] A. Alshnakat, D. Gurov, C. Lidström, and P. Rümmer, "Constraint-based contract inference for deductive verification," in *20 Years of KeY*. Springer, 2020, vol. 12345, pp. 149–176.
- [46] A. Albarghouthi, I. Dillig, and A. Gurfinkel, "Maximal specification synthesis," in *POPL*. ACM, 2016, pp. 789–801.
- [47] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S. Cheung, and C. Tian, "Enchanting program specification synthesis by large language models using static analysis and program verification," in *CAV (2)*, vol. 14682. Springer, 2024, pp. 302–328.
- [48] S. A. Seshia, "Combining induction, deduction, and structure for verification and synthesis," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2036–2051, 2015.
- [49] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Informatica*, vol. 54, no. 7, pp. 693–726, 2017.
- [50] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik, "Synthesizing environment invariants for modular hardware verification," in *VMCAI*, vol. 11990. Springer, 2020, pp. 202–225.

APPENDIX

APPENDIX A

LARGE LANGUAGE MODEL (LLM) PROMPTING

We use chain-of-thought prompting to produce pre/post-condition contracts for procedures. The prompt is structured into three stages: (1) task description and instruction, (2) procedure behavior reasoning, and (3) contract generation.

In the first stage, the task is introduced and the requirements are specified. At this point, a Python-based DSL is also provided so that the LLM can employ it in the later contract-generation stage.

```
1 Your task is to create two Boolean expressions,
2 a precondition and a postcondition, for a function
3 named '<function_name>'. Instructions below:
4 1. Precondition should only depend on inputs.
5 2. Postcondition can depend on inputs and outputs.
6 3. Express pre/post-conditions using the following
7    set of Python DSL.
8
9 # Addition operation
10 def Add(a, b): return a + b
11 ...
```

Next, the LLM performs a code reasoning step. It is instructed to identify: (1) the assumptions the function might make prior to execution, and (2) the guarantees it provides afterward. When refining an existing contract, the model is further asked to specify (3) additional constraints that distinguish between a given set of positive and negative examples. In this refinement setting, we also prompt the LLM to explain why the previous contract was invalid and to propose corrective modifications. We observe that explicitly reasoning about such additional constraints enables the model to refine contracts more effectively in light of the provided examples.

```
1 Describe what the function
2 1. may assume before it is executed, and
3 2. guarantees after it is executed. Also,
4 3. think about additional constraints that would
5    satisfy (eliminate) positive (negative) examples.
6 Output a list of the above points and nothing else.
7
8 The function has the following inputs and outputs:
9 Inputs: <inputs>; Outputs: <outputs>
10 Code below: <code>
11
12 Here is a cex of the following pre/postconditions:
13 Precondition: <precondition>
14 Postcondition: <postcondition>
15 Counterexample: <cex>
16 Write an explanation of why the pre/postconditions
17 are violated in the form
18 ```json
19 {
20   "explanation": <EXPLANATION>
21   "suggestion": <SUGGESTION>
22 }
23 ```
24
25 Lastly, below are some input/output examples.
26 Make sure the conditions satisfy the positive
27 examples and dissatisfy the negative examples.
28 Compare the positive and negative examples and
29 deduce what makes the negative examples invalid
30 behaviors of the code.
31 List constraints that need to be added to avoid
```

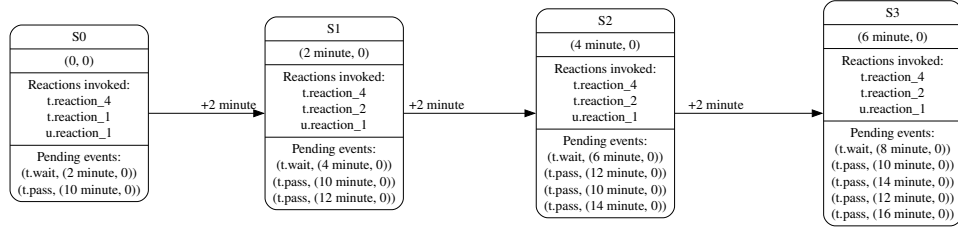
```
32 the negative examples.
33
34 Positive example 1: ...
35 ...
36 Negative example 1: ...
37 ...
```

After completing the reasoning step, the LLM is instructed to generate the contract in the provided Python DSL, adhering to the specified requirements. Any SUGGESTION extracted from its earlier analysis is highlighted in the prompt to guide this generation. A Python template is then supplied for the model to complete, along with an example to serve as a usage reference. The final pre/post-conditions are extracted by parsing the Python code block returned in this stage.

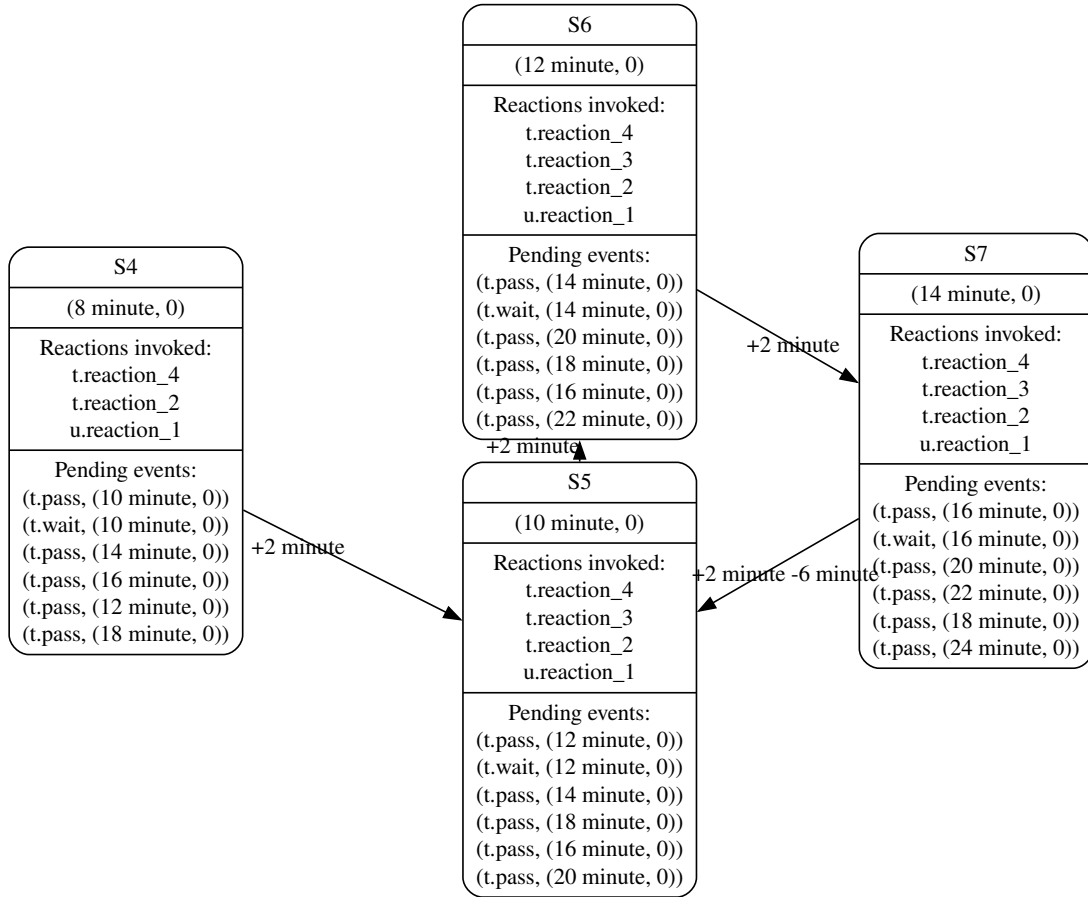
```
1 Based on your analysis, complete the two Python
2 functions PRECONDITION and POSTCONDITION that
3 returns pre/postconditions for the code.
4 Requirements:
5 1. The function body should be a single return
6    statement. Do not use intermediate variables.
7 2. Link all fields of the output to the inputs
8    if possible, even if they are not used.
9 3. You may abstract the postcondition by ignoring
10    details and focusing on the relationship
11    between the inputs and outputs.
12 ...
13
14 **NOTE**: <SUGGESTION> Focus on this when generating
15 pre/postconditions.
16
17 Complete the code below using the provided
18 Python DSL and put it in a Python code block.
19 ```python
20 def PRECONDITION(query):
21     return # TODO: Fill in using the provided DSL
22 def POSTCONDITION(query, response):
23     return # TODO: Fill in using the provided DSL
24 ```
25
26 Below is an example of using the DSL.
27 ```python
28 def PRECONDITION(in):
29     return TrueExpr()
30 def POSTCONDITION(in, out):
31     return And(
32         Eq(Select(out, 'count'), Select(in, 'count')),
33         Eq(Select(out, 'is_present'), TrueExpr()),
34     )
35 ```
```

APPENDIX B

STATE SPACE DIAGRAM FOR LF PROGRAM VERIFICATION



(a) The initialization phase of SSD (including S4)



(b) The periodic phase of SSD (excluding S4)

Fig. 6: The State Space Diagram (SSD) of the LF program in Figure 5. The SSD is a lasso, with an initialization phase in Figure 6a and a periodic phase in Figure 6b.

Quantifying Robustness of Medical Image Segmentation Networks Using TensorStars

Meghan Stuart

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, United States of America
mstuart@cs.unc.edu

Parasara Sridhar Duggirala 

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, United States of America
psd@cs.unc.edu

Abstract—This paper presents a framework for quantifying robustness of U-Nets, a widely used neural network architecture in 3 dimensional biomedical image segmentation. We first present a metric for measuring robustness of biomedical image segmentation methods. We then propose TensorStars, a new representation to model the perturbation of a given medical image and overapproximate the effect of these perturbations on the image segmentation performed by a 3 dimensional U-Net. Given the large dimensionality of medical images, naive implementation of overapproximation results in poor scalability of robustness verification. We propose two main improvements, (1) leveraging hardware acceleration from GPUs for performing tensor operations, and (2) leverage the CPU parallelism for accelerating the ReLU overapproximation. We observe that these two enhancements improve the runtime by 10x when implemented on a 32 core CPU with NVidia GeForce RTX with 8GB VRam. We perform robustness analysis on a U-Net that performs segmentation of kidneys with cross entropy loss of less than 0.01. Our evaluation demonstrates that perturbing less than 3% of input voxels can result in misclassification of up to 20% of voxels in the segmentation output. To the best of authors knowledge, this is the first work that extends the robustness verification of neural networks to biomedical imaging data.

I. INTRODUCTION

Inspired by the success of deep learning in various computer vision tasks such as classification and segmentation, researchers have extended these deep learning techniques to medical image analysis tasks such as classification, segmentation, and registration. Currently, neural networks are the state-of-the-art in performing most of the medical image analysis tasks. These neural networks are currently integrated into various workflow for treatment. For example, deep learning is frequently used and studied in the radiotherapy treatment workflow, to perform automatic segmentation of at-risk organs [1], [2], [3], for registration of images taken during multi-day treatments [4], [5], [6], for image enhancement like multi-modality image fusion [7], [8] or metal artifact reduction [9], [10], lesion detection and characterization [11], [12], landmark detection for fiducial-free motion tracking [13], and even medical image synthesis [14], [15] which must be used carefully to ensure proper model generalizability [16]. Given the safety critical nature of such integrating such neural networks into clinical procedures, one has to ensure that the outcomes of the neural networks deployed in biomedical images are robust. In order for deep learning technology to be integrated into the clinical workflow,

clinicians must have confidence in the tool. They expect model transparency and interpretability, and require assurances of the model's ability to perform consistently through rigorous quality assurance testing during the initial implementation of the technology, as well as on an ongoing basis to ensure its adaptability to evolving imaging techniques and treatment protocols [17].

Empirical evaluation of neural networks in medical imaging and beyond, does not inspire such confidence. It has been shown that deep learning models, in general, lack generalizability to out of distribution samples [18]. In the specific domain of medical imaging, it has been shown in [19] that a model trained for pneumonia detection on lung radiographs on data from a specific hospital may have significantly poorer performance on the same diagnostic task using data from a different hospital. This may be due to differences in imaging technology, image processing or diagnostic protocols. *To provide confidence and explainability of neural networks used in medical imaging, in this paper, we verify the robustness of U-Nets, a widely used neural network architecture in segmentation. That is, given a set of perturbations (possibly uncountable), does the segmentation outcome of the U-Net change as a result of the perturbations.*

Unique Aspects of Biomedical Image Segmentation and its effects on Robustness Verification: There are significant differences between deep learning techniques used in biomedical image analysis when compared to disciplines such as computer vision or natural language processing. First, biomedical images are typically 3 dimensional in nature and it requires significant effort to generate high quality labeled data for segmentation and other downstream tasks. Second, the scale of data is significant smaller, that is, datasets often contain 30-50 patient data and data augmentation techniques play a key role in generating labeled data. Finally, the widely used architecture for 3 dimensional data, namely U-Nets have a unique feature called skipped-connection layers that are not commonly used in other disciplines. These differences require a careful examination of formulating and verifying robustness properties of biomedical image segmentation networks.

A lot of prior work on neural network verification was targeted towards demonstrating robustness of specific neural networks such as ACASXu [20] or VGG-16 [21]. However, in contrast, we do not verify a standard neural network available

on a public domain as that is not the modality of usage of neural networks in medical imaging domain. In fact, researchers in medical image analysis discourage using the same network for different data sets as each network is trained on data from a small number of patients.

Therefore, to assess the robustness of a biomedical image segmentation networks we propose the following workflow. First, we pick a widely used neural network architecture and training algorithm in the literature and train a new neural network on a publicly available data set. Next, we formulate important robustness properties over biomedical images and verify these properties on the trained neural network.

The authors understand the circularity in this argument. If the neural network is poorly chosen, then it would be unsurprising to find violations of robustness property. Our answer to this circularity is two-fold. (1) the architecture, training algorithm, and loss functions picked by us are widely accepted to be standard in the medical imaging literature and are now part of several commercial products, and (2) our main contribution is to not to demonstrate that a specific network violates or satisfies the properties, but rather the algorithmic procedure can indeed verify the properties of neural networks used in medical image analysis.

Another key difference between neural networks for segmentation and classification is the scale of the output. A segmentation network is required to assign a class for each pixel in the image, as opposed to a classification network that assigns a single class to the entire image. In the case of biomedical image segmentation, each voxel is classified to be either an *organ* or a *background* voxel. Therefore, verifying robustness of a segmentation network on a 32x32x32 biomedical image is equivalent to verifying the robustness of 32^3 or 32,768 individual classifications. In this paper, instead of constructing individual verification instances, we construct an overapproximation of the classification results on all the voxels and efficiently verify the robustness of the segmentation network.

Contributions of Our Paper: In this paper we propose a new representation called TensorStar for modeling perturbations of a given medical image. We show that noise models specific to medical imaging and noise models widely used in neural network verification can be modeled as TensorStars. We then propose algorithms for constructing a conservative overapproximation of the segmentation output of a U-Net when given inputs in the TensorStar. We demonstrate that U-Nets for segmenting Kidneys on a given data set are not robust with respect to perturbations. Our verification demonstrates that while minor perturbations of the image does not lead to any change in segmentation outcome, changing merely 3% of voxels can change the segmentation of Kidney by 20%. **To the best of authors knowledge, this is the first paper on robustness verification for neural networks in medical imaging domain.**

II. PROBLEM FORMULATION

We consider 3D images consisting of voxels. Each voxel is associated with a specific intensity recorded from the sensor. We model the space of images as $\mathbb{R}^{n_1 \times n_2 \times n_3}$ where n_1 , n_2 , and n_3 define the resolution of the image in x , y , and z dimensions. Typically the intensity of an image lies in the interval $[0, 255]$.

In this particular paper, we focus on segmentation networks. For the sake of simplicity, we focus on single object segmentation. That is, there is only one organ of interest in the image, and the segmentation network identifies whether a given voxel belongs to the organ or the background. Given an image I is an element of $\mathbb{R}^{n_1 \times n_2 \times n_3}$, a segmentation network $\mathcal{N} : \mathbb{R}^{n_1 \times n_2 \times n_3} \rightarrow \mathbb{B}^{n_1 \times n_2 \times n_3}$ where each voxel is tagged with a boolean value. A voxel is labelled as 1 if it is classified to be a part of the segment (organ) and 0 if it is classified to be part of the background. In general, medical images have same resolution in the two axial directions ($n_1 = n_2$). In research models, the images are often sampled such that the dimensions are the same in all three dimensions, i.e., $n_1 = n_2 = n_3$.

Our goal is to verify the robustness property of segmentation networks. That is, given a set of perturbations of a given image (possibly uncountable), would the classification of voxels change from background to segment and vice versa. Given an set of input images $\mathcal{I} \subset \mathbb{R}^{n_1 \times n_2 \times n_3}$, the set of possible segmentation outputs $\mathcal{O} = \bigcup_{I \in \mathcal{I}} \mathcal{N}(I)$. Given a reference image I_{ref} , a voxel $[i, j, k]$ is said to be fragile if it is classified into different classes in the reference and the perturbations. That is, $\exists I \in \mathcal{I}$ such that $\mathcal{N}(I)[i, j, k] \neq \mathcal{N}(I_{\text{ref}})[i, j, k]$. Given an set $\mathcal{I}$, we define the segmentation fragility as the fraction of voxels that are classified as fragile when compared to the voxels of interest, i.e., the voxels that are classified as segment.

$$\text{fragility} = \frac{\#\text{fragile voxels}}{\#\text{voxels of interest}}.$$

While it might be convenient to count the ratio of fragile voxels to total number of voxels, we propose this definition to better account for signal-to-noise ratio in biomedical image segmentation. Naturally, a fragility of 0 would mean that none of the voxels have fragile segmentation and a fragility of 1 would mean that the noise induced by perturbing the input is equal to that of the size of the segment. Also observe that under increased perturbation, fragility of a set could be more than 1.

We use reachable set computation techniques for computing overapproximation of the image of the classification network on the set of perturbed inputs. Therefore, we might not be able to exactly compute the fragility of a given segmentation network on the set $\mathcal{I}$. We therefore compute underapproximation and overapproximation of the network fragility.

III. BIOMEDICAL IMAGE SEGMENTATION USING U-NETS

One of the widely used neural architectures in medical imaging is U-Net. The U-Net architecture was introduced by [22] in 2015 specifically for biomedical image analysis. The architectural diagram as introduced in the seminal paper is shown in Figure 1. It is somewhat similar in form to a

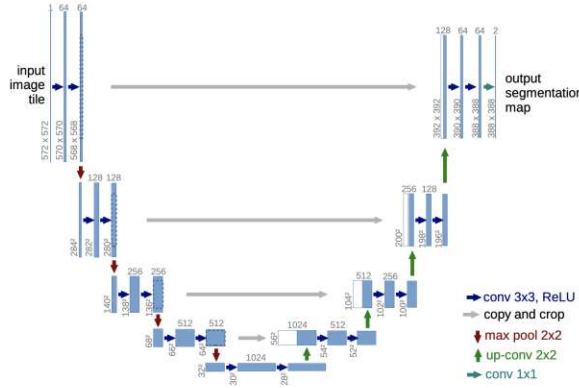


Fig. 1: Original U-Net architecture for biomedical image segmentation as introduced in [22].

convolutional autoencoder, in that the U-Net first performs convolutional downsampling for feature detection like an encoder, then convolutional upsampling to synthesis new information from the detected features like a decoder. However, the U-Net also uses “skip connections” where output from intermediate layer on the left side of the U are saved and provided to layers on the right side of the U in order to provide important higher resolution spatial context for the model to use in its synthesis. It has been shown empirically that U-Nets are significantly better than other architectures comprising of purely convolutional components. While readers might be familiar with most of the components, we briefly explain the components of the U-Net architecture primarily because these deal with 3 dimensional inputs.

Downsampling Convolutions (Conv): A 3 dimensional convolutional kernel is an array of dimensions $l \times w \times d$ denoted as $K[l][w][d]$. Given an input of dimensions $n_1 \times n_2 \times n_3$, $I[n_1][n_2][n_3]$, the convolution of $I * K$ is an array of dimensions $(n_1 - l) \times (n_2 - w) \times (n_3 - d)$ and the element defined as

$$(I * K)[i, j, k] = \sum_{\substack{p \in [0, l-1], \\ q \in [0, w-1], \\ r \in [0, d-1]}} I[i + p, j + q, k + r] \cdot K[p, q, r].$$

For generalization, one can add bias that will be added while doing weighted sum during convolution.

Rectified Linear Unit (ReLU): The rectified linear unit acts as a filter that removes all the values less than 0. In an U-Net, a ReLU layer is performed after the convolution operation. Given an input of dimensions $n_1 \times n_2 \times n_3$, the ReLU operation is another tensor of size $n_1 \times n_2 \times n_3$ defined as

$$\text{ReLU}(I)[i, j, k] = \begin{cases} 0 & \text{if } I[i, j, k] \leq 0. \\ I[i, j, k] & \text{otherwise.} \end{cases}$$

Skipped Connection: The output tensor of a specific downsampling convolution is saved, concatenated with the output from a later upsampling convolution, and given as input to subsequent convolutional layers.

Upsampling Convolution (ConvTranspose): A 3 dimensional upsampling convolution kernel is an array of size $l \times w \times d$ denoted by $K[l][w][d]$. Given an input I of dimensions $n_1 \times n_2 \times n_3$, $I[n_1][n_2][n_3]$, an upsampling convolution (ConvTranspose) $I ** K$ is an array of size $(n_1 \cdot l) \times (n_2 \cdot w) \times (n_3 \cdot d)$ as defined as

$$(I ** K)[l \cdot i + p, w \cdot j + q, d \cdot k + r] = I[i, j, k] \cdot K[p, q, r].$$

where $i \in [0, n_1 - 1]$, $j \in [0, n_2 - 1]$, $k \in [0, n_3 - 1]$ and $p \in [0, l - 1]$, $q \in [0, w - 1]$, $r \in [0, d - 1]$. Informally, each voxel of the image I is magnified by a factor of $l \times w \times d$, weighted by the factor defined in the kernel. The ConvTranspose is sometimes presented as convolution of an image after introducing new voxels — whose number is defined by padding and stride — with zero values and then performing the usual convolution operation. It is easy to observe that such variants are mathematically equivalent to a sequence of one more convolution operations that can be described in simple mathematical terms. It is a usual practice that after such upsampling, one would often perform convolutions to smoothen the image.

While in the original U-Net paper, one suggested using MaxPooling layer, empirical results have shown that one could achieve accuracy using U-Nets without MaxPool layer. The last layer in a U-Net is a softmax layer where each voxel is compared to a threshold and is classified as segment if the value is more than threshold and is classified as background if the value is less than the threshold.

IV. VERIFICATION OF IMAGE-NET NEURAL NETWORKS USING TENSORSTARS

In this section, we present the extension of star sets to 3 dimensional images and call them TensorStars. We introduce the definition of TensorStars, present the operations that overapproximate the Conv, ReLU, Skip, ConvTrans operations on TensorStars, and present an algorithm to compute fragility of a given perturbation for a reference.

Definition 1: A TensorStar of dimensions $n_1 \times n_2 \times n_3$ is a tuple $\mathcal{T} = \langle a, G, P \rangle$ where a is a 3 dimensional array of dimensions $n_1 \times n_2 \times n_3$, $G = \{g_1, g_2, \dots, g_m\}$ is a set of m generators where each g_i is a 3 dimensional array of dimensions $n_1 \times n_2 \times n_3$ and P is a predicate $P : \mathbb{R}^m \rightarrow \{\top, \perp\}$. The set of tensors defined by a Tensor star is denoted as

$$[\mathcal{T}] = \{ I \mid \exists \alpha_1, \dots, \alpha_m, I = a + \alpha_1 g_1 + \dots + \alpha_m g_m \text{ and } P(\alpha_1, \dots, \alpha_m) = \top \}.$$

Informally, a TensorStar is an affine combination of anchor and generator tensor where the coefficient of the anchor is 1 and the coefficients for the generators satisfy the predicate in the TensorStar. We abuse notation and use $\mathcal{T}$ to represent both the tuple representation of a TensorStar and the set represented by the tuple.

In general, there is no restriction on the format of each of the anchors and the generators. However, if one wants to model

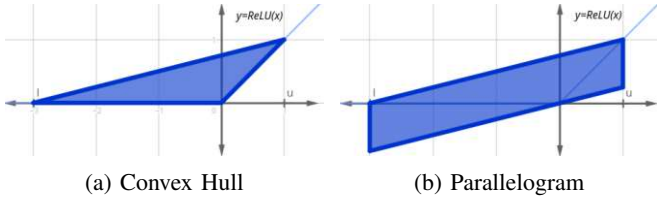


Fig. 2: Overapproximation of a ReLU on a single voxel

voxel level perturbation of an image, each g_i is represented as a 3 dimensional array with all zeros except for one voxel. Similarly, while the predicate P can be in general any predicate, efficient solvers exist only for only a restricted subset. We often consider the predicate P to be defined as a conjunction of linear constraints.

One of the primary advantages of using the star representation is that performing affine transformations over the set represented by the star can be computed very easily by applying the affine transformation over the anchor and generators. The predicate that captures the complex dependency between the coefficients remains unchanged during affine transformations and needs only to be modified during nonlinear operations such as ReLU .

A. Operations on Tensor Stars

Convolution and Transpose Convolution Operation: Given a convolution kernel K of dimensions $l \times w \times d$, the convolution operation on a TensorStar $\mathcal{T} = \langle a, G, P \rangle$, denoted as $\mathcal{T} * K = \langle a * K, G * K, P \rangle$ where $G * K = \{g_1 * K, \dots, g_m * K\}$. That is, for computing convolution over TensorStars, we perform convolution operations on anchor and generators.

Lemma 1: Given a TensorStar $\mathcal{T}$ and a convolutional kernel K , $\llbracket \mathcal{T} * K \rrbracket = \{I * K \mid I \in \llbracket \mathcal{T} \rrbracket\}$. That is, the convolution operation on anchor and generators is equivalent to performing convolution on all the tensors represented by TensorStar.

Proof: Follows easily from the definition of Conv and the affine nature of convolution operation. ■

Similar to the convolution operation, given an upsampling convolution kernel K of dimensions $l \times w \times d$, the upsampling convolution operation on a TensorStar $\mathcal{T} = \langle a, G, P \rangle$, denoted as $\mathcal{T} ** K = \langle a ** K, G ** K, P \rangle$ where $G ** K = \{g_1 ** K, \dots, g_m ** K\}$. The correctness of the transformation is very similar to the convolution operation.

ReLU Operation: Several previous works have overapproximated the nonlinear ReLU operation as convex operation. Two of the techniques presented in the literature are the minimal

convex hull and parallelogram overapproximation. The minimal convex hull is given in Figures 2a and the parallelogram convex hull is given in Figure 2b.

In medical imaging, since ReLU operation is applied for each voxel, we first collect the constraints that determine the possible values assigned for a voxel. That is, for a given voxel $[i, j, k]$ the possible range of values in $\mathcal{T}$ for voxel $[i, j, k]$ is given by the constraints:

$$\mathcal{T}[i, j, k] = \{v \mid v = a[i, j, k] + \sum_{l=1 \dots m} \alpha_l g_l[i, j, k] \wedge P(\bar{\alpha}) = \top\}$$

Where $\bar{\alpha}$ is the vector $(\alpha_1, \dots, \alpha_m)$. Since P is a conjunction of linear constraints, one can compute upper and lower bounds of $\mathcal{T}[i, j, k]$ using linear programming and we represent the upper and lower bounds as $u_{[i, j, k]}$ and $l_{[i, j, k]}$. Once computed, we overapproximate the region as a parallelogope or minimal convex hull. We give the example as a parallelogope by introducing a new variable β and introduce the constraints as follows $\beta \geq 0$; $\beta \geq \mathcal{T}[i, j, k]$; and $\beta \leq \frac{u_{[i, j, k]} \mathcal{T}[i, j, k]}{u_{[i, j, k]} - l_{[i, j, k]}}$. We illustrate this through an example of such transformation over 3 dimensional tensors.

Example 1: Consider the TensorStar as given in Figure 3a where the tensor star consists of an anchor and only one generator and the predicate is defined by constraints $\alpha \in [-2, 2]$. All the values in the anchor and generator that are behind the visible voxels are 0. For the first voxel (highlighted in blue), the upper and lower bound is given as $\max\{-1 + \alpha \mid \alpha \in [-2, 2]\}$ and $\min\{-1 + \alpha \mid \alpha \in [-2, 2]\}$. Therefore $u = 1$; $l = -3$. Since the upper and lower bounds have intersection with values both below and above 0, we change the corresponding index in the anchor tensor and make it 0. Further, since after the ReLU transformation we model the values in the particular voxel using the variable β , we make the corresponding voxel value of the first generator to be 0. We then add the additional constraints on β as $\beta \geq 0$; $\beta \geq -1 + \alpha$; and $\beta \leq \frac{1((-1+\alpha)-(-3))}{1-(-3)}$. The last constraint can be rewritten as $\beta \leq \frac{\alpha+2}{4}$.

In this specific example, only one of the voxel values has a nonempty intersection with values greater than and less than 0. Therefore, we only introduce one new variable. In theory, all the voxels in the TensorStar can potentially have nonempty intersection with values greater than and less than 0. In such cases, we have to introduce $n_1 \times n_2 \times n_3$ new variables and add thrice the number of constraints — that can clearly cause an explosion in the number of constraints and variables. However, in practice, only a few voxels are capable of achieving both

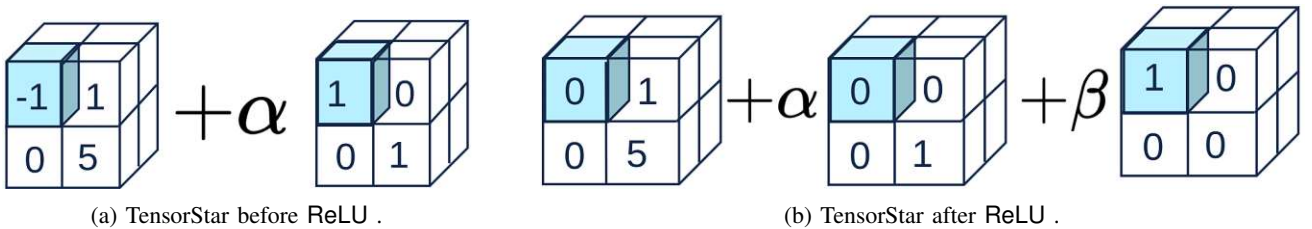


Fig. 3: Overapproximation of a ReLU on a TensorStar.

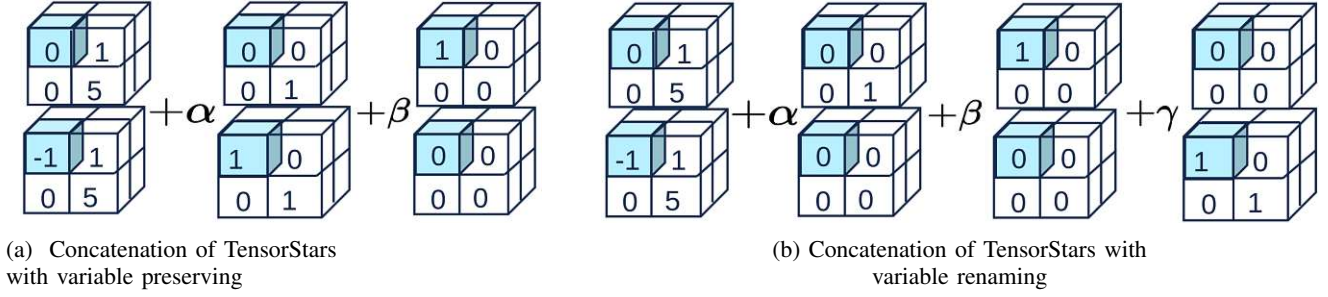


Fig. 4: Concatenation Operation from Skip for TensorStars.

negative and positive values and therefore we do not suffer from the explosion of such variables.

Lemma 2: Minimal Convex Hull and Parallelogram overapproximations of the TensorStar compute the overapproximation of ReLU operation on the set of tensors represented by the TensorStar.

Concatenation From Skipped Connection Operation: In a U-Net, the values of voxels after a convolution layer are concatenated together with the voxels obtained after upsampling. In theory, one can concatenate two 3 dimensional images of any sizes where one would add sufficient padding around the smaller image and make sure that both the images have compatible dimensions. In practice, we rarely develop such U-Nets and instead make sure that the dimensions of the images being concatenated is exactly the same.

Observe that the TensorStar obtained after the upsampling has been transformed through several ReLU layers as opposed to the TensorStar from the skipped connection. Therefore, the TensorStar from the upsampling can potentially have more variables — and therefore, more generators. This begs the question, how to concatenate two tensor stars that do not have the same number of generators.

There are two possible ways of concatenating such TensorStars. In the first technique, we preserve the mapping from the generator to the corresponding coefficient and concatenate the generators for the same coefficient. The second technique is to forget the link between the generators and the corresponding coefficients and instead generate new set of variables. Since a ReLU operation only creates and adds new variables, the original constraints on the older variables remain.

Example 2: Consider the two TensorStars represented in Figure 3a and Figure 3b. The two ways to concatenate the TensorStars are given in Figure 4. In Figure 7a, we preserve the relationship between the coefficient variable α and concatenate the two corresponding generators. Since the first tensor star does not have a corresponding generator for the coefficient β , the corresponding generator is concatenated with 0s. In Figure 7c, we do not preserve the relationship between the generators. Instead, we create a new variable γ and copy the constraints into a new variable.

Consider two TensorStar $T_1 = \langle a_1, G_1, P_1 \rangle$ and $T_2 = \langle a_2, G_2, P_2 \rangle$ where T_1 is from the skipped connection with $G_1 = \{g_1, g_2, \dots, g_{k_1}\}$ and $G_2 = \{g'_1, g'_2, \dots, g'_{k_2}\}$ where G_1

and G_2 are determined by the order of the coefficient variables with $k_2 \geq k_1$ and all the constraints in P_1 are also part of P_2 .

The variable preserving way to concatenate T_1 and T_2 is given as $T_1 \odot_p T_2 = \langle a_1 \odot a_2, G_1 \odot_c G_2, P_2 \rangle$ where $G_1 \odot_c G_2 = \{g_1 \odot g'_1, g_2 \odot g'_2, \dots, g_{k_1} \odot g'_{k_1}, \mathbf{0} \odot g'_{k_1+1}, \dots, \mathbf{0} \odot g'_{k_2}\}$. Here $\odot$ operation is overloaded to be concatenation operation on 3 dimensional arrays. That is, we concatenate the generators corresponding to the same coefficient variables and for all the coefficient variables that do not have the corresponding generators in T_1 , we concatenate $\mathbf{0}$.

The variable renaming way to concatenate T_1 and T_2 is given as $T_1 \odot_r T_2 = \langle a_1 \odot a_2, G_1 \odot_r G_2, P'_1 \odot_r P_2 \rangle$ where $G_1 \odot_r G_2 = \{g_1 \odot \mathbf{0}, \dots, g_{k_1} \odot \mathbf{0}, \mathbf{0} \odot g'_1, \dots, \mathbf{0} \odot g'_{k_2}\}$ and $P'_1 \odot_r P_2$ renames all the variables in P_1 to fresh variables. Notice that variable renaming way would increase the number of generators to be $k_1 + k_2$ as opposed to k_2 in the variable preserving way. While it would seem obvious that one would want to reduce the number of variables, in some instances, the number of generators with all 0 values becomes large. Therefore, for improving the efficiency of 3 dimensional array multiplication, we drop the generators with all zero coefficients and use variable renaming.

Lemma 3: Given TensorStars T_1 and T_2 , the variable preserving and variable renaming concatenation denoted as $T_1 \odot_p T_2$ and $T_1 \odot_r T_2$ compute the overapproximation of the concatenation from Skip in the U-Net.

Robustness Verification of U-Nets using TensorStars: The algorithm for computing an overapproximation of the input TensorStar when operated on by a neural network is provided in Algorithm 1. The algorithm applies the corresponding transformation on the TensorStar as presented earlier in this section. After computing the overapproximation of the a TensorStar with the five different components, we finally get the TensorStar representation before the softmax layer. For each voxel, we compute the upper and lower bound of the threshold values for the voxel and compare them with the threshold. If both the upper and lower bound are above the threshold, we assign that the voxel is classified as the segment; similar for both the values below the threshold. If the upper bound of the voxel is above the threshold and the lower bound is below the threshold, we label the corresponding voxel as unstable.

Algorithm 1: Algorithm that computes an overapproximation of the output when the given input TensorStar is operated on by the input Neural Network.

input : Neural Network: $\mathcal{N}$ obtained by sequential composition of components $\mathcal{N}_1, \dots, \mathcal{N}_k$; Input TensorStar $\mathcal{I}$.

output : A TensorStar $\mathcal{O}$ that is an overapproximation of $\mathcal{N}(\mathcal{I})$.

```

1  $\mathcal{O}_0 \leftarrow \mathcal{I}$ ;
2 for each  $i$  from 1 to  $k$  do
3   if  $\mathcal{N}_i$  is Conv then
4      $\mathcal{O}_i \leftarrow \mathcal{O}_{i-1} * \mathcal{N}_i$ ;
      // Convolution on TensorStar
5   if  $\mathcal{N}_i$  is ConvTranspose then
6      $\mathcal{O}_i \leftarrow \mathcal{O}_{i-1} * \mathcal{N}_i$ ;
      // Upsampling Convolution
7   if  $\mathcal{N}_i$  is ReLU then
8      $\mathcal{O}_i \leftarrow \text{ReLU}(\mathcal{O}_{i-1})$ ;
      // ReLU on TensorStar
9   if  $\mathcal{N}_i$  is SkipConnection then
10    Create a copy of  $\mathcal{O}_{i-1}$ ;
11     $\mathcal{O}_i \leftarrow \mathcal{O}_{i-1}$ ;
      // Skip connections stores a
      // copy that will be used during
      // concatenation.
12  if  $\mathcal{N}_i$  is Concatenation of  $\mathcal{N}_{i-1}$  and  $\mathcal{N}_j$  then
13     $\mathcal{O}_i \leftarrow \mathcal{O}_{i-1} \odot \mathcal{O}_j$ ;
      // Concatenation with or without
      // variable renaming as determined
      // by the user performing
      // verification.
14 return  $\mathcal{O}_k$ ;

```

V. LEVERAGING ACCELERATORS AND PARALLELISM FOR IMPROVED EFFICIENCY OF TENSOR STAR VERIFICATION

There are two main inefficiencies in the overapproximation of TensorStars for the neural network layers. First, the typical size of modern medical image is greater $128 \times 128 \times 128$. In this study, we downsample images to $32 \times 32 \times 32$. Therefore, each convolution operation would have to perform 32^3 3 dimensional array multiplication operations for each anchor and generator in the TensorStar. Similarly, the ReLU operation is performed on each and every voxel after every convolutional layer, therefore, the ReLU overapproximation would have to perform approximately $32^3 \approx 40,000$ linear programs for computing the upper and lower bounds. Given the large number of these operations, any fractional improvement in the performance of these two operations, would have a large impact on the overall performance of the robustness verification. *We have implemented two performance enhancements and as a result, have achieved 10x improvement in the time taken for verification.*

GPU Acceleration for Tensor Multiplications: One of the primary reasons for wide-scale usage of Graphical Processing Units for machine learning is their efficiency at performing tensor products. All purely affine transformations were accelerated on the GPU using TensorFlow’s Keras backend [23].

Parallelization of ReLU Overapproximation: At each voxel v , determining the overapproximate reachable region of the ReLU involves solving two linear programs determined by the TensorStar predicate to find u , and l , the upper and lower bounds of the set reachable at v . These linear programs can be solved independently for each voxel, hence can profit from program parallelism. This is called leveraging the *embarassing parallelism* in the algorithm.

We accelerated the ReLU computations by integrating python’s multiprocessing library [24]. Naive implementation in the multiprocessing library would cause a lot of busy-waiting on concurrent data structures and do not improve the performance. To overcome this, we save all the constraints on coefficient variables, anchors, and generators in a shared memory. Then each individual thread will read the corresponding coefficients for each ReLU and formulate a local linear program that invokes the python linear programming library. The results from each individual linear program is written to a thread specific write locations, avoiding the usage of global lock on concurrent data structure. This is represented diagrammatically in Figure 5. This acceleration achieves a greater than $10\times$ speed up in sample verification instances, as shown in table I.

I	A	Param	AT (s)	RT (s)
35	D	(240, 0.005)	2630	20113
35	D	(240, 0.015)	2923	21372
35	D	(180, 0.005)	3099	39324
35	D	(180, 0.010)	7062	136539
35	P	(18, 10, 12) - 1 - [100,102]	2723	22598
36	D	(240, 0.005)	2639	21215
36	D	(240, 0.015)	2704	22547
36	P	(18, 10, 12) - 1 - [100,102]	2731	24153
37	D	(240, 0.005)	2659	22018
37	D	(240, 0.015)	3764	32637
37	P	(18, 10, 12) - 1 - [100,102]	2723	22270
Average Speedup			10.79	

TABLE I: Speed up using accelerators for neural network verification. Legend: **I**: Reference Image ID; **A**: Perturbation type D - Darkening, B - Beam Hardening, P - Patch; textbfParam: parameters for perturbation — Threshold and ϵ for darkening, min and max intensity and the layers for beam hardening, location, patch size of $n \times n \times n$ and intensity range for patch; **RT**: running time without acceleration in seconds; **AT**: running time with acceleration in seconds

VI. EVALUATION

To demonstrate the effectiveness of the robustness verification using TensorStars, we verify the robustness of a U-Net performing Kidney segmentation.

A. Training of Kidney Segmentation Network

As described in Section I, in the domain of medical image segmentation, there is a lot more emphasis on the architecture

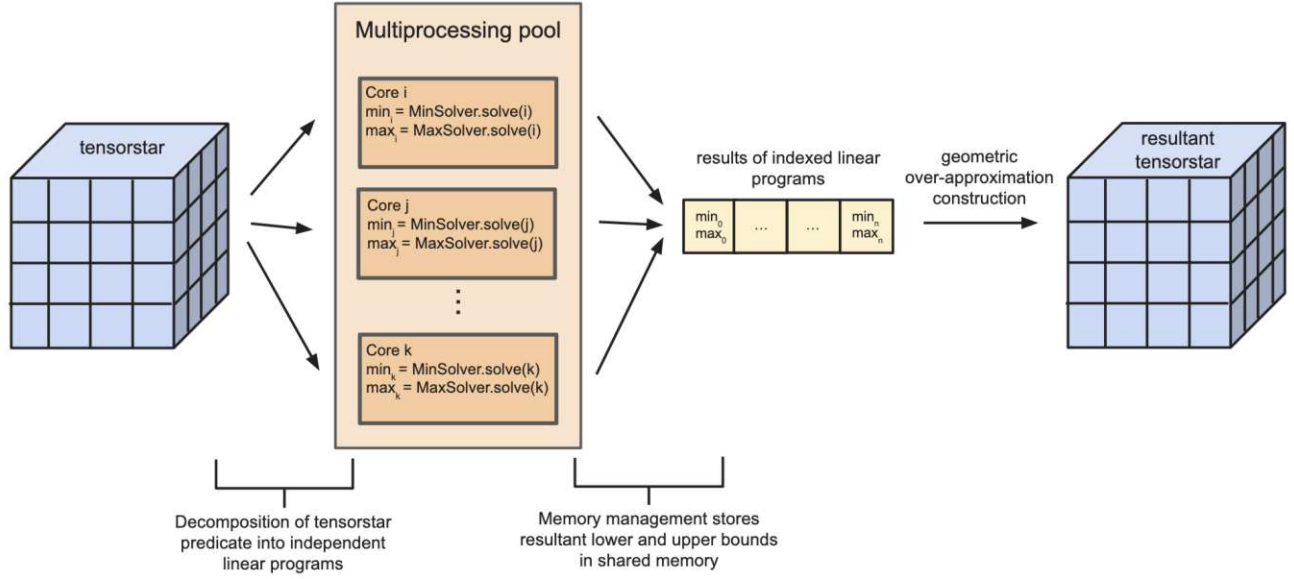


Fig. 5: Multiprocessor acceleration of ReLU overapproximation computation. MinSolver and MaxSolver are cvxpy based linear program instances giving the minimum and maximum, respectively, achievable value of a given voxel.

and training technique as opposed to the specific network used in segmentation. Therefore, we trained our own Kidney segmentation network with the 3D UNet as described in [25] implemented in Python 3.7 with TensorFlow2, with minor modifications; specifically, restricting to ReLU activation functions and removing pooling for simplicity. The model was trained on 30 randomly selected cases from the KITS datasets [26] with the standard data supplementation techniques. While the standard image sizes are $128 \times 128 \times 128$, we downsampled image sizes to $32 \times 32 \times 32$ to make the training and verification more efficient. The total number of resulting ReLU operations in this network are $\approx 460,000$. We used categorical cross entropy for loss function and used the ADAM optimizer trained for 1000 epochs on a machine with Intel Core i7-11700F Rocket Lake 2.5GHz (4.9GHz Turbo) 8-Core 16-Thread CPU and NVIDIA GeForce RTX 3060 Ti 8GB GPU. The final value of loss function was less than 0.001.

B. Perturbation of Input Using Noise Models

We consider three main types of perturbations on a given input: *darkening*, *beam hardening*, and *patch*.

Darkening Perturbation: For this perturbation, we select an intensity threshold T and for all the voxels of intensity greater than T , the intensity is reduced by a factor of $[0, \epsilon]$ where $\epsilon \in \{0.005, 0.01, 0.015\}$. That is, if a given voxel has intensity of 200 with a threshold of $T = 180$ and $\epsilon = 0.005$, then the intensity of the voxel in the TensorStar is restricted to be in the interval $[0, 0.005 \times 200] = [0, 1]$. Given a reference image, this perturbation can be modeled as a tensor star where the anchor is the reference image except for the voxels with intensity above T made 0. For each voxel erased from the anchor, we add a corresponding generator where only that voxel is given a value 1 and all other voxels are 0. The coefficient of the

corresponding generator is the range of allowed intensities defined by the darkening factor ϵ .

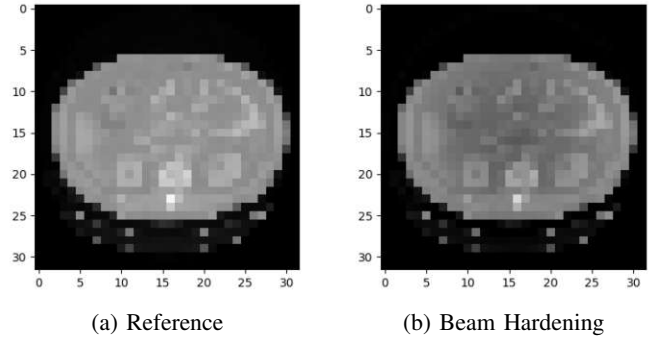


Fig. 6: An example of Beam Hardening with decrease in intensity at the center by 100 units.

Beam Hardening Perturbation: Biomedical researchers have observed several distortions in the intensity of x-rays when they pass through human tissue as early as 1970s [27]. One such widely documented noise is Beam Hardening and more specifically cupping effect where low intensity beams fail to reflect from the center of the image. As a result, the values of intensity appears to be lower at the center of the image but remain unchanged at the edges of the image. While computational techniques to account for beam hardening have been known in the literature [28], [29], we intend to check whether a given neural network is robust with respect to beam hardening perturbation of a given intensity. To model such cupping effect, we introduce a maximum and minimum decrease in the intensity of the center of the image $[min_c, max_c]$ and the gradient is uniform such that the edges of the image have no loss in intensity. Given a reference image, this perturbation can be modeled with the reference image

I	A	# K	# U	# M	F	# IG	# FG	Param	RT
35	D	168	0	1	0.006	1	18	(240,0.005)	0:44
35	D	168	0	1	0.006	1	57	(240,0.01)	0:44
35	D	168	0	1	0.006	1	85	(240,0.015)	0:49
35	D	166	0	5	0.03	4	149	(200,0.005)	0:57
35	D	166	0	5	0.03	4	285	(200,0.01)	2:45
35	D	166	0	5	0.03	4	450	(200,0.015)	7:58
35	D	165	1	5	0.03	6	156	(180,0.005)	0:52
35	D	165	1	5	0.03	6	344	(180,0.01)	1:58
35	D	165	1	5	0.03	6	502	(180,0.015)	4:22
35	D	158	1	41	0.25	50	1016	(128,0.005)	35:20
35	D	-	-	-	-	-	-	(128,0.01)	TO
35	B	158	1	18	0.11	1	821	(50,51) - [12,13]	18:01
35	B	-	-	-	-	-	-	(50,150) - [8,12]	OOM
35	P	169	0	2	0.006	1	53	(18,10,12) - 1 - [100,102]	0:45
35	P	173	3	26	0.17	3	613	(18,10,12) - 3 - [100,102]	17:38
35	P	-	-	-	-	5	-	(18,10,12) - 5 - [100,102]	TO
36	D	151	0	0	0.0	1	33	(240,0.005)	0:44
36	D	151	0	0	0.0	1	48	(240,0.01)	0:44
36	D	151	0	0	0.0	1	57	(240,0.015)	0:45
36	D	148	0	5	0.03	6	208	(200,0.005)	1:17
36	D	148	0	5	0.03	6	337	(200,0.01)	3:52
36	D	148	0	5	0.03	6	518	(200,0.015)	14:47
36	B	125	1	27	0.19	1	1278	(50,51) - [12,13]	83:31
36	B	-	-	-	-	-	-	(50,150) - [8,12]	OOM
36	P	152	0	1	0.006	1	58	(18,10,12) - 1 - [100,102]	0:46
36	P	126	3	39	0.28	3	1118	(18,10,12) - 3 - [100,102]	159:07
36	P	-	-	-	-	5	-	(18,10,12) - 5 - [100,102]	TO
37	D	77	0	1	0.012	2	43	(240,0.005)	0:44
37	D	77	0	1	0.012	2	74	(240,0.01)	0:49
37	D	77	0	1	0.012	2	107	(240,0.015)	1:03
37	D	78	0	2	0.024	5	113	(200,0.005)	1:00
37	D	78	1	2	0.024	5	194	(200,0.01)	2:02
37	D	78	0	2	0.024	5	296	(200,0.015)	5:52
37	B	75	0	25	0.33	1	837	(50,51) - [12,13]	15:12
37	B	-	-	-	-	-	-	(50,150) - [8,12]	OOM
37	P	78	0	2	0.024	1	44	(18,10,12) - 1 - [100,102]	0:46
37	P	74	1	14	0.20	3	787	(18,10,12) - 3 - [100,102]	41:40
37	P	-	-	-	-	5	-	(18,10,12) - 5 - [100,102]	TO

TABLE II: Legend: **I**: Reference Image ID; **A**: Perturbation type D - Darkening, B - Beam Hardening, P - Patch; **# K**: # of voxels classified as Kidney in reference image; **# U**: # of unstable voxels; **# M**: # of provably misclassified voxels; **F**: Fragility of the network on perturbation; **# IG**: # of generators in the initial TensorStar; **# FG**: # of generators in the final TensorStar; **Param**: parameters for perturbation — Threshold and ϵ for darkening, min and max intensity and the layers for beam hardening, location, patch size of $n \times n \times n$ and intensity range for patch; **RT**: running time in hrs:min;

as the anchor, and one generator where the coefficients of generator models the uncertainty in the decrease of intensity in the interval $[min_c, max_c]$ and the corresponding generator is a uniform gradient with coefficient 1 in the center that gradually tapers to 0 at the edges of the image. An example of one layer of an image and its corresponding beam hardening perturbation is given in Figure 6.

Patch Perturbation: In this perturbation, we select a patch of voxels of a specific size, say 3×3 , and modify the intensity in these voxels to be determined by a specific range. Similar to darkening perturbation, patch perturbation can also be modeled using TensorStars.

C. Verification Results

After computing the overapproximation of TensorStar with U-Net, we compute the number of guaranteed correctly classified voxels, number of unstable voxels, and number of guaranteed misclassified voxels. All of our experiments have been performed on an Ubuntu 22.04 system with 32 Core CPU (AMD Ryzen Threadripper PRO 7975WX Processor (5.3 GHz,

32 Cores)) and 256 GB main memory with NVIDIA Geforce RTX 4090 with 24GB VRAM Memory. The results of the experiments on different types of perturbations and the time taken for such verification is provided in Table II. Examples of images with unstable, misclassified, and correctly classified voxels is given in Figure 7 (in Appendix). Since the actual images are 3 dimensional, the visual representation of all the guaranteed misclassified and unstable voxels is available here ¹.

We present some observations from the verification results. First is that the amount of time for verification would definitely depend on the specific reference image and the type of perturbation. Second, the darkening perturbation is the easiest to verify whereas the beam hardening and patch are often computationally intensive. In some instances, the fragility of the neural network is around 0.2 and in our experiments going beyond this value requires significant computational resources. Third, while some of the misclassification of voxels is happening at the places where perturbation is introduced,

¹<https://tinyurl.com/h8mpj9t8>

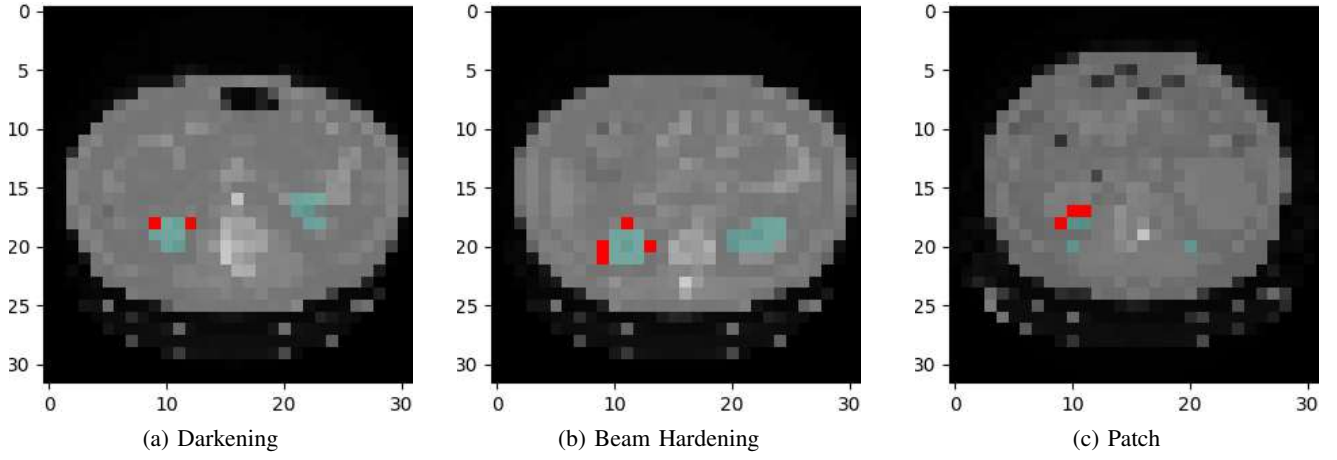


Fig. 7: Medical images with provably incorrectly classified voxels labelled in red and the Kidney labelled in green.

in several instances, we observed misclassification in layers that are directly not perturbed. This is specifically in the case of Beam Hardening where introducing uncertainty in only one layer introduced misclassification in multiple layers of the image. Finally, while there are instances where we could not finish verification in a week (TO: for time out in the results table), we encountered a new bottleneck for verification. In various beam hardening perturbations we realized that the system ran out of memory (OOM: system ran out of memory and crashed in results table). This could be possibly because of two reasons. First, as the size of the perturbations and constraints grew, the memory required for storing the TensorStar in shared memory also grew. Second, the memory requirements for solving the linear programs also grew with the size of the linear program.

D. Comparison with Other Tools and Verification Benchmarks

Techniques for verification of neural networks can be broadly classified into one of two classes: constraint satisfaction driven or representation driven. Examples of constraint satisfaction driven works are Reluplex [20], β -Crown [30], $\alpha - \beta$ Crown [31], Marabou [32], Sherlock [33], Planet [34], etc. In these works, the goal is to formulate the robustness verification as a constraint solving problem and develop general decision procedures for accelerating the search for robustness violation or satisfaction. In the representation driven techniques for robustness verification, the goal is to compute an explicit representation of possible outcomes of a given input set. The types of representations used vary from Zonotopes, to Polyhedra, to Star sets. Examples of tools that use this technique are ERAN [35], [36], DeepPoly [37], NNV [38], and verification using ImageStars [21], and others [39], [40]. For a full review of current state-of-the-art methods of deep learning model verification see [41]. All of the verification tools currently use VNN-LIB [42] for specifying properties.

While in theory the tools for verifying robustness properties of neural networks can work with input of any dimensions, in practice these tools support limited verification instances.

First, the representation based techniques do not support 3 dimensional input data and skipped connections features that are prominent in U-Nets. Despite our best efforts, we could not use $\alpha - \beta$ Crown for verifying robustness of 3 dimensional U-Net and have received confirmation from the developers that they have not tested the $\alpha - \beta$ Crown on 3 dimensional inputs. Additionally, beam-forming, a widely documented source of noise in medical imaging cannot be represented using the current VNN-LIB standard. This is because beam forming requires correlating the perturbations of different voxels using a common variable. Hence, our work fills an important and necessary gap in the domain of robustness verification of neural networks.

VII. CONCLUSION AND FUTURE WORK

While the current work falls in the category of robustness verification using representation, it differs from the literature in three key aspects. First, none of these works have been specifically tailored to U-Nets. The specific three dimensional nature of the input and output modalities require special purpose implementation that is currently not achieved by other representation based tools. Second, due to the high dimensionality of medical imaging data, the number of ReLU operations encountered in these verifications is at least an order of magnitude larger than neural networks for 2D images trained on CIFAR dataset. Without leveraging the parallelism and hardware acceleration, verifying robustness of medical imaging networks would be unachievable. Finally, unlike purely synthetic perturbations of input images, we demonstrated that verification techniques can be used to verify real noise profiles such as Beam Hardening.

To the best of authors knowledge, this is the first paper on verifying robustness of neural networks on medical imaging data. We have shown that extension of star based techniques, together with enhancements can indeed verify robustness of realistic perturbations that can be observed in practice. While we have demonstrated the robustness (or the lack thereof) for U-Nets, the size of the U-Nets considered in this paper are

not fully applied in clinical setting. Our goal is to scale the verification to high resolution medical images of $512 \times 512 \times 512$ which would require improving the efficiency by at least three orders of magnitude.

Acknowledgements: The authors would like to acknowledge the support from AFOSR under award number FA9550-23-1-0286 and NSF under awards 2038960 and 2303564. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR or NSF.

REFERENCES

- [1] Nima Tajbakhsh, Laura Jeyaseelan, Qian Li, Jeffrey N. Chiang, Zhihao Wu, and Xiaowei Ding. Embracing imperfect datasets: A review of deep learning solutions for medical image segmentation. *Medical Image Analysis*, 63:101693, July 2020.
- [2] Fang Lu, Fa Wu, Peijun Hu, Zhiyi Peng, and Dexing Kong. Automatic 3D liver location and segmentation via convolutional neural network and graph cut. *International Journal of Computer Assisted Radiology and Surgery*, 12(2):171–182, February 2017.
- [3] Fang Liu, Zhaoye Zhou, Hyungseok Jang, Alexey Samsonov, Gengyan Zhao, and Richard Kijowski. Deep convolutional neural network and 3D deformable approach for tissue segmentation in musculoskeletal magnetic resonance imaging. *Magnetic Resonance in Medicine*, 79(4):2379–2391, April 2018.
- [4] Yabo Fu, Yang Lei, Tonghe Wang, Kristin Higgins, Jeffrey D. Bradley, Walter J. Curran, Tian Liu, and Xiaofeng Yang. LungRegNet: An unsupervised deformable image registration method for 4D-CT lung. *Medical Physics*, 47(4):1763–1774, April 2020.
- [5] Xiao Yang, Roland Kwitt, Martin Styner, and Marc Niethammer. Quicksilver: Fast predictive image registration – A deep learning approach. *NeuroImage*, 158:378–396, September 2017.
- [6] Boah Kim, Dong Hwan Kim, Seong Ho Park, Jieun Kim, June-Goo Lee, and Jong Chul Ye. CycleMorph: Cycle consistent unsupervised deformable image registration. *Medical Image Analysis*, 71:102036, July 2021.
- [7] Tongxue Zhou, Su Ruan, and Stéphane Canu. A review: Deep learning for medical image segmentation using multi-modality fusion. *Array*, 3-4:100004, September 2019.
- [8] Yi Li, Junli Zhao, Zhihan Lv, and Jinhua Li. Medical image fusion method by deep learning. *International Journal of Cognitive Computing in Engineering*, 2:21–29, June 2021.
- [9] Lequan Yu, Zhicheng Zhang, Xiaomeng Li, and Lei Xing. Deep Sinogram Completion With Image Prior for Metal Artifact Reduction in CT Images. *IEEE Transactions on Medical Imaging*, 40(1):228–238, 2021.
- [10] Haofu Liao, Wei-An Lin, S. Kevin Zhou, and Jiebo Luo. ADN: Artifact Disentanglement Network for Unsupervised Metal Artifact Reduction. *IEEE Transactions on Medical Imaging*, 39(3):634–643, March 2020.
- [11] P. Herent, B. Schmauch, P. Jehanno, O. Dehaene, C. Saillard, C. Balleyguier, J. Arfi-Rouche, and S. Jégou. Detection and characterization of MRI breast lesions using deep learning. *Diagnostic and Interventional Imaging*, 100(4):219–225, April 2019.
- [12] Ke Yan, Youbao Tang, Yifan Peng, Veit Sandfort, Mohammadhadi Bagheri, Zhiyong Lu, and Ronald M. Summers. MULAN: Multitask Universal Lesion Analysis Network for Joint Lesion Detection, Tagging, and Segmentation. In Dinggang Shen, Tianming Liu, Terry M. Peters, Lawrence H. Staib, Caroline Essert, Sean Zhou, Pew-Thian Yap, and Ali Khan, editors, *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019*, pages 194–202, Cham, 2019. Springer International Publishing.
- [13] W. Zhao, D.P. Capaldi, C. Chuang, and L. Xing. Fiducial-Free Image-Guided Spinal Stereotactic Radiosurgery Enabled Via Deep Learning. *International Journal of Radiation Oncology*Biophysics*, 108(3):e357, November 2020.
- [14] Jean-Francois Rajotte, Robert Bergen, David L. Buckeridge, Khaled El Emam, Raymond Ng, and Elissa Strome. Synthetic data as an enabler for machine learning applications in medicine. *iScience*, 25(11):105331, November 2022.
- [15] Jelmer M. Wolterink, Anna M. Dinkla, Mark H. F. Savenije, Peter R. Seevinck, Cornelis A. T. Van Den Berg, and Ivana Išgum. Deep MR to CT Synthesis Using Unpaired Data. In Sotirios A. Tsaftaris, Ali Gooya, Alejandro F. Frangi, and Jerry L. Prince, editors, *Simulation and Synthesis in Medical Imaging*, volume 10557, pages 14–23. Springer International Publishing, Cham, 2017. Series Title: Lecture Notes in Computer Science.
- [16] Ahmed Alaa, Boris Van Breugel, Evgeny S. Saveliev, and Mihaela van der Schaar. How Faithful is your Synthetic Data? Sample-level Metrics for Evaluating and Auditing Generative Models. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 290–306. PMLR, July 2022.
- [17] Liesbeth Vandewinckele, Michaël Claessens, Anna Dinkla, Charlotte Brouwer, Wouter Crijns, Dirk Verellen, and Wouter Van Elmpt. Overview of artificial intelligence-based applications in radiotherapy: Recommendations for implementation and quality assurance. *Radiotherapy and Oncology*, 153:55–66, December 2020.
- [18] Avital Oliver, Augustus Odena, Colin A Raffel, Ekin Dogus Cubuk, and Ian Goodfellow. Realistic Evaluation of Deep Semi-Supervised Learning Algorithms. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- [19] John R. Zech, Marcus A. Badgeley, Manway Liu, Anthony B. Costa, Joseph J. Titano, and Eric Karl Oermann. Variable generalization performance of a deep learning model to detect pneumonia in chest radiographs: A cross-sectional study. *PLOS Medicine*, 15(11):e1002683, November 2018.
- [20] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In Rupak Majumdar and Viktor Kunčák, editors, *Computer Aided Verification*, volume 10426, pages 97–117. Springer International Publishing, Cham, 2017.
- [21] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, volume 12224, pages 18–42. Springer International Publishing, Cham, 2020.
- [22] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, volume 9351, pages 234–241. Springer International Publishing, Cham, 2015.
- [23] François Chollet et al. Keras. <https://keras.io>, 2015.
- [24] Python. multiprocessing — process-based parallelism — python 3.8.3rc1 documentation.
- [25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.
- [26] Nicholas Heller, Fabian Isensee, Klaus H Maier-Hein, Xiaoshuai Hou, Chunmei Xie, Fengyi Li, Yang Nan, Guangrui Mu, Zhiyong Lin, Miofei Han, et al. The state of the art in kidney and kidney tumor segmentation in contrast-enhanced ct imaging: Results of the kits19 challenge. *Medical image analysis*, 67:101821, 2021.
- [27] Rodney A Brooks and Giovanni Di Chiro. Beam hardening in x-ray reconstructive tomography. *Physics in medicine & biology*, 21(3):390, 1976.
- [28] Gabor T Herman. Correction for beam hardening in computed tomography. *Physics in Medicine & Biology*, 24(1):81, 1979.
- [29] Elke Van de Castele, Dirk Van Dyck, Jan Sijbers, and Erik Raman. An energy-based beam hardening model in tomography. *Physics in Medicine & Biology*, 47(23):4181, 2002.
- [30] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.
- [31] Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. General cutting planes for bound-

propagation-based neural network verification. *Advances in Neural Information Processing Systems*, 2022.

- [32] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, volume 11561, pages 443–452. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.
- [33] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output Range Analysis for Deep Feedforward Neural Networks. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods*, volume 10811, pages 121–138. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.
- [34] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*, pages 269–286. Springer, 2017.
- [35] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting Robustness Certification of Neural Networks. In *International Conference on Learning Representations (ICLR)*. 2019.
- [36] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. PRIMA: Precise and General Neural Network Certification via Multi-Neuron Convex Relaxations, 2021. [_eprint: 2103.03638](#).
- [37] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, January 2019.
- [38] Hoang-Dung Tran, Xiaodong Yang, Diego Manzananas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, volume 12224, pages 3–17. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.
- [39] Timon Gehr, Matthew Mirman, Dana Drachler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, San Francisco, CA, May 2018. IEEE.
- [40] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the Single Neuron Convex Barrier for Neural Network Certification. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.
- [41] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.
- [42] Stefano Demarchi, Dario Guidotti, Luca Pulina, Armando Tacchella, Nina Narodytska, Guy Amir, Guy Katz, and Omri Isac. Supporting standardization of neural networks verification with vnnlib and coconet. In *FoMLAS@ CAV*, pages 47–58, 2023.

Of Good Demons and Bad Angels: Guaranteeing Safe Control under Finite Precision

^{1st} Samuel Teuber 
Karlsruhe Institute of Technology
Karlsruhe, Germany
teuber@kit.edu

^{2nd} Debasmita Lohar 
Karlsruhe Institute of Technology
Karlsruhe, Germany
debasmita.lohar@kit.edu

^{3rd} Bernhard Beckert 
Karlsruhe Institute of Technology
Karlsruhe, Germany
beckert@kit.edu

Abstract—As neural networks (NNs) become increasingly prevalent in safety-critical neural network-controlled cyber-physical systems (NNCSs), formally guaranteeing their safety becomes crucial. For these systems, safety must be ensured throughout their entire operation, necessitating infinite-time horizon verification. To verify the infinite-time horizon safety of NNCSs, recent approaches leverage Differential Dynamic Logic ($d\mathcal{L}$). However, these $d\mathcal{L}$ -based guarantees rely on idealized, real-valued NN semantics and fail to account for roundoff errors introduced by finite-precision implementations.

This paper bridges the gap between theoretical guarantees and real-world implementations by incorporating robustness under finite-precision perturbations—in sensing, actuation, and computation—into the safety verification. We model the problem as a *hybrid game* between a *good Demon*, responsible for control actions, and a *bad Angel*, introducing perturbations. This formulation enables formal proofs of robustness w.r.t. a given (bounded) perturbation. Leveraging this bound, we employ state-of-the-art mixed-precision fixed-point tuners to synthesize sound and efficient implementations, thus providing a complete end-to-end solution. We evaluate our approach on case studies from the automotive and aeronautics domains, producing efficient NN implementations with rigorous infinite-time horizon safety guarantees.

Index Terms—Differential Dynamic Logic, Mixed-Precision Fixed-Point Tuning, Neurally-Controlled Systems

I. INTRODUCTION

Neural networks (NNs) are increasingly integrated into critical systems, such as adaptive cruise control in cars [1]–[3] or collision avoidance in airplanes [4], [5]. Thus, ensuring the safety of NN Control Systems (NNCSs) becomes imperative.

Recent research has made significant progress toward this goal. Some approaches [6]–[19] only analyze the input-output behavior of the NN but ignore the essential feedback-loop dynamics between the NN and its physical environment. Others [20]–[31] consider this feedback loop but only verify safety for a *finite*-time horizon, which may be insufficient. A recent technique, VerSAILLE [32], addresses this limitation by proving safety throughout the entire operation (*infinite-time horizon*) of the NNCS using Differential Dynamic Logic ($d\mathcal{L}$).

However, VerSAILLE’s strong guarantees rely on idealized, real-valued NN semantics and do not account for roundoff errors arising from practical finite-precision (floating- or fixed-point) implementations. As recently also demonstrated empirically [33], finite-precision arithmetic introduces errors at

potentially every operation, which can accumulate and lead to incorrect or unsafe decisions [34]. Furthermore, efficiency demands often require implementations in low-precision fixed-point arithmetic [35], [36], introducing larger numerical errors in the computation. Thus, to ensure real-world safety, it is crucial to bridge this implementation gap [37] by verifying that safety guarantees are valid under finite-precision semantics.

While verification of quantized NNs has been explored [38], it is known to be PSPACE-hard and typically focuses on input-output specifications rather than closed-loop dynamics, which must simultaneously account for the continuous, real-valued behavior of physical dynamics. Sound quantization techniques [39] automate precision tuning but leave the burden of safety verification to the user. A recent work [40] has combined safety analysis with bounded roundoff errors, but their guarantees are limited to finite-time horizons.

This leaves a critical gap: the need for an efficient and practical method to guarantee *infinite-time horizon safety* of NNCSs while accounting for finite-precision errors.

In this paper, we propose a general infinite-horizon safety verification methodology, building on VerSAILLE [32], that accounts for finite-precision errors in sensing, actuation, and computation. We observe that some “safe” $d\mathcal{L}$ control strategies (*envelopes*) can become unsafe even under miniscule input or output perturbations, making them unsuitable for verifying realistic (NN) controllers. To address this, we introduce a new notion of *robustness* for control envelopes, which is also of independent interest beyond handling roundoff errors. Finally, we integrate our approach with mixed-precision tuning to generate sound and efficient fixed-point implementations that can be directly compiled using standard high-level synthesis tools for FPGAs, providing a complete end-to-end solution. To summarize, we make the following contributions:

- (C1) We formalize robustness for control envelopes under perturbation – a necessary condition for finite-precision $d\mathcal{L}$ -based NN verification – using Differential Game Logic ($d\mathcal{GL}$), as a hybrid game between a *good Demon* (control actions) and a *bad Angel* (perturbations). We present a decidable, sufficient criterion for robustness.
- (C2) We propose a novel, decidable criterion for guaranteeing the infinite-time horizon safety of NNs under given perturbations leveraging existing *real-valued* NN verifiers.

This work was supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF).

- (C3) We integrate precision tuning (with Daisy [41]) to synthesize efficient mixed-precision fixed-point implementations that guarantee a provided error bound and are compatible with standard hardware synthesis tools [42].

We evaluated our approach on three realistic case studies: an Adaptive Cruise Control (ACC) system [1], [32] with discrete and continuous acceleration control, and a Vertical Airborne Collision Avoidance System (VCAS) [4]. Our analysis shows that the continuous ACC control envelope from [32] is not robust to output perturbations; we propose modifications and verify safety under bounded errors. For VCAS, our results imply (limited) robustness of the original control envelope. Using the computed perturbation bounds, we synthesized mixed-precision fixed-point implementations for all case studies, compiled them with Xilinx Vivado [42], and measured their latency (running time) in machine cycles. An extended version of this paper (including the Appendix) can be found on arXiv [43].

II. APPROACH AT A GLANCE

This section provides a high-level overview using a simplified ground robot from [45]. The technical background and the proposed approach are presented in the subsequent sections.

a) Ground Robot Example:

The robot, at distance $p \in \mathbb{R}$ to an obstacle, moves along a corridor (see Figure 2). It controls velocity v , which can be updated once per control cycle of duration of at most T . To ensure safety, the robot must maintain $p \geq 0$.

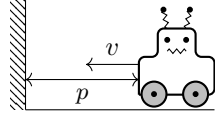


Fig. 2. A robot at distance p controls its velocity v and avoids an obstacle ($p \geq 0$).

Let us consider the case where the robot can only move toward the wall (i.e., $v \geq 0$). Since the robot has full control over its velocity, a safe strategy is to choose any $v \geq 0$ such that $p - Tv \geq 0$, ensuring it does not collide with the obstacle within one control cycle. The aim is to formally prove that the robot remains safe under this strategy.

b) *Nomenclature:* As we discuss control strategies at different levels, we unify the terminology as follows: First, nondeterministic control strategies modeled and verified in $d\mathcal{L}$ are referred to as *control envelopes*. Second, concrete real-valued NNs that deterministically provide a control action for a given input are called *idealized implementations*, and will be verified against control envelopes. Finally, hardware-level realizations of idealized implementations using fixed-point arithmetic are termed *fixed-point implementations*. We will bound their numerical errors w.r.t. their idealized counterparts.

c) *Verifying Control under Finite Precision:* A control envelope, where v is chosen arbitrarily such that $p - Tv \geq 0 \wedge v \geq 0$, can be proven safe in real-valued $d\mathcal{L}$.

In practice, real-valued arithmetic does not exist, and finite precision introduces pitfalls: When p is close to 0, even infinitesimal perturbations to v can cause $p < 0$, leading to a collision. Braking earlier *cannot* resolve this, as similar perturbations could, by induction, still push the robot into an unsafe region. This illustrates a *non-robust* control envelope,

highlighting the need to *rethink* the system structure—or at least the model—by, e.g., also admitting negative velocities.

d) *Our Approach:* Our work extends $d\mathcal{L}$ -based safety guarantees to finite-precision NNs while: 1) still providing guarantees w.r.t. precise real-valued physical dynamics modeled in $d\mathcal{L}$ and 2) leveraging efficient, state-of-the-art real-valued NN verification tools. We build on VerSAILLE [32], which proves real arithmetic refinement relations between NNs and provably safe $d\mathcal{L}$ control envelopes. Just as VerSAILLE, and as indicated in Figure 1, our approach assumes the availability of a safe $d\mathcal{L}$ model of the analyzed control system. In particular, this $d\mathcal{L}$ model comprises a (nondeterministic) *control envelope* which encodes provably safe control actions. To consider perturbations arising from sensing, computation (e.g., fixed-point arithmetic), or actuation, we model such perturbations as real-valued hybrid programs, termed as *angelic perturbations*. For example, in our robot example, we define an angelic perturbation α_{noise} executed after the original control envelope chooses v : It nondeterministically selects a bounded perturbation $|\varepsilon_v| \leq \delta_v$ and then updates v via $v := v + \varepsilon_v$. This captures bounded perturbation of up to δ_v for the robot's actuation, whether due to numerical imprecision or actuator inaccuracies.

As can be seen in Figure 1, these angelic perturbations can be used to multiple ends: First, we can use them to analyze the robustness of a given $d\mathcal{L}$ model/control envelope. To this end, we provide a game-based robustness definition and decidable sufficient criteria for robustness. Second, for a robust control envelope, we extend VerSAILLE's analysis to take account of angelic perturbations in a decidable manner. This allows us to prove infinite-time horizon NNCS safety under perturbation. Finally, for NNs that are safe under perturbation, we can leverage sound quantization techniques, which synthesize a fixed-point implementation of the real-valued NN where the maximal error bound matches our angelic perturbation.

Towards our contribution (C1), we provide a game-theoretic definition for the robustness of control envelopes: There must *exist* control actions that keep the system safe irrespective of the environment's *angelic perturbations*. Classic $d\mathcal{L}$ safety properties ($[\alpha] \psi$) interpreted as game show that Demon, who has no influence on execution, has a winning strategy irrespective of Angel's control and environment choices. Robustness hands over the controller choices to a *good* Demon who must choose actions to weather a *bad* Angel adversarially choosing perturbations and environment behavior. Our formulation induces *decidable* real-arithmetic conditions applicable to *any* discrete, loop-free angelic perturbation. We identify *liveness* (a control envelope does not get *stuck*), as a special case of robustness. For our robot, a control envelope with $v \geq 0$ is not robust under α_{noise} , but an envelope with bidirectional movement is (choose any v such that $p - T(v + \delta) \geq 0$).

Concerning (C2), we extend VerSAILLE to account for angelic perturbations: NN verification reduces to checking whether the NN implements a *winning strategy* for the robustness game formulated in (C1) and robustness becomes a *necessary* condition for $d\mathcal{L}$ -based NN verification under finite

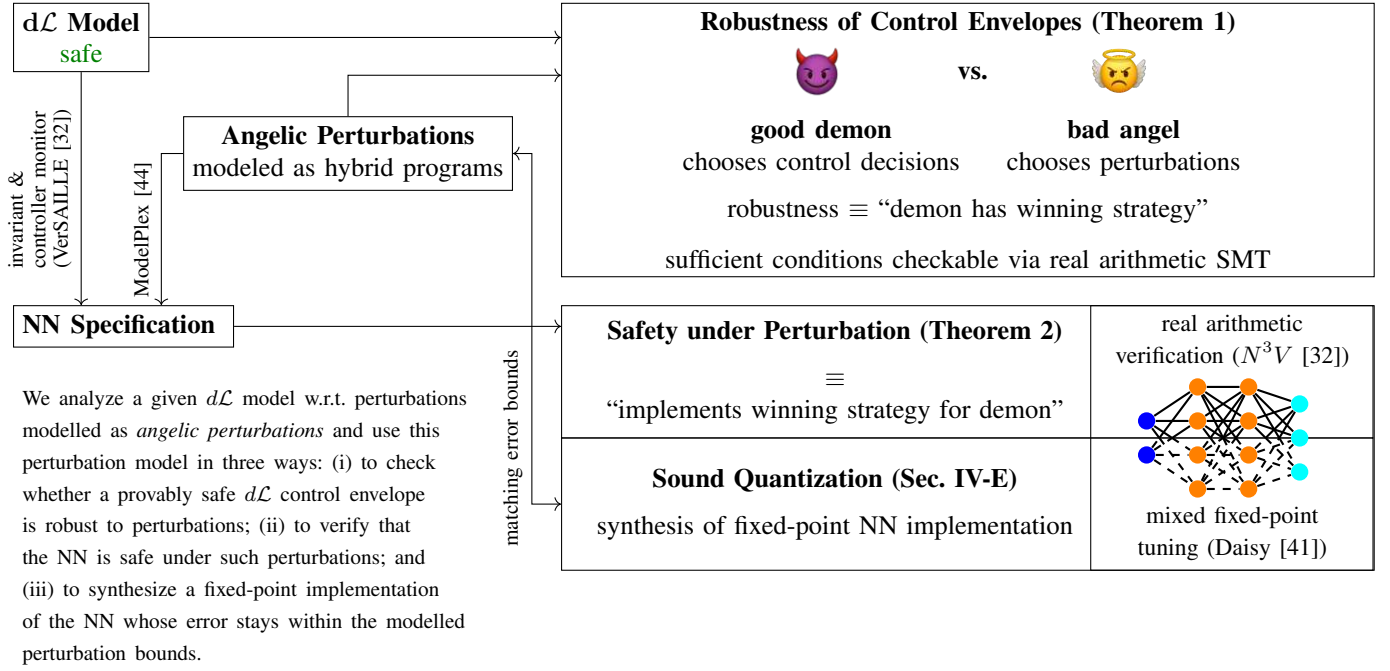


Fig. 1. Overview of approach which assumes a safe $d\mathcal{L}$ model as foundation (emojis generated by ChatGPT)

precision. For the robot with bidirectional movement, the real-valued condition $p \geq 0 \rightarrow p + Tf(p) \geq 0$, where $f(p)$ is the velocity computed by the NN f can be extended to $-\delta_v \leq \varepsilon_v \leq \delta_v \wedge p \geq 0 \rightarrow p + T(f(p) + \varepsilon_v) \geq 0$, which is also compatible with existing linear and nonlinear NN verification tools. Disproving NN safety amounts to finding a concrete input p and perturbation ε_v violating the specification. Proving the specification yields an *infinite-time horizon safety guarantee* for the NN control system under bounded perturbations. Our approach systematically handles a broad class of perturbations with decidable construction from $d\mathcal{L}$ control envelopes.

Addressing (C3), we leverage NN safety guarantees under bounded perturbations and utilize Daisy for mixed-precision fixed-point tuning, with added pre- and post-processing steps to support typical NN structures and operations, enabling direct hardware synthesis using Xilinx Vivado [42].

Thus, our *end-to-end* approach ensures infinite-horizon safety under bounded finite-precision errors and produces sound, efficient, hardware-deployable NN controllers. While in this paper, we focus on NNs due to current tool support, the approach is broadly applicable and can be straightforwardly extended to other types of controllers.

III. BACKGROUND

This section provides background on proving infinite-time horizon safety for NN Control Systems using $d\mathcal{L}$ (Sec. III-A) and on efficient code generation for finite-precision (Sec. III-B). We illustrate these concepts using the robot example from Figure 2.

A. Safety Verification

We review Differential Dynamic Logic [46] for abstractly reasoning about the hybrid (continuous and discrete) behavior of Cyber-Physical Systems, Differential Game Logic [47] for handling adversarial dynamics, and describe the $d\mathcal{L}$ -based NN verification technology VerSAILLE [32].

1) *Differential Dynamic Logic*: Differential Dynamic Logic ($d\mathcal{L}$) [46], [48]–[50] is a (first-order) dynamic logic [51], [52], i.e., a modal logic over a state space $\mathcal{S}$ whose modalities are parametrized with programs. The syntax of *hybrid programs* analyzed in $d\mathcal{L}$ is given by the following grammar, where x is a variable, Q is a real arithmetic formula, e is either $*$ or a real arithmetic term, and $f(x)$ is a real arithmetic function over x (with α and β representing hybrid programs):

$$\alpha, \beta ::= \underbrace{x' = f(x) \ \& \ Q}_{\text{continuous}} \mid \underbrace{x := e \mid ?(Q) \mid \alpha; \beta \mid \alpha \cup \beta}_{\text{discrete and loop-free}} \mid \underbrace{(\alpha)^*}_{\text{loop}}$$

The semantics of hybrid programs are described as state transitions where a state ν is a map from variables x to (real) values $\nu(x)$. For example, for e a real arithmetic term, the semantics of variable assignment are defined as $\llbracket x := e \rrbracket = \{(\nu, \mu) \in \mathcal{S}^2 \mid \mu = \nu_x^e\}$ where ν_x^e is a state that evaluates x to e and all other variables y to $\nu(y)$.

The defining feature of hybrid programs is their ability to model *continuous* dynamics: $x' = f(x) \ \& \ Q$ denotes a program where x nondeterministically evolves along the given differential equation within an evolution constraint Q , allowing x to take *any* reachable value satisfying Q . The primitive $x := *$ represents a nondeterministic assignment. The remaining constructs are: predicate checks (no transition if the check fails), sequential composition, nondeterministic choice, and looping. Safety properties can be expressed as formulas $P \rightarrow [\alpha] Q$,

stating that, if the precondition P holds, then Q holds after all (nondeterministic) executions of the hybrid program α . For example, the formula $x = 2 \rightarrow [y := x \cup y := x + x] y > 0$ is valid because for any state with $x = 2$, all possible executions ensure that $y > 0$ (specifically $y \in \{2, 4\}$) after execution.

$d\mathcal{L}$ also provides a dual modality to express liveness: $P \rightarrow \langle \alpha \rangle Q$ guarantees that, starting from any state satisfying P , it is *possible* to reach *some* state where Q holds. Moreover, the equivalence $[\alpha] Q \equiv \neg \langle \alpha \rangle \neg Q$ applies.

We also use the subset of *concrete, discrete, and loop-free* programs, corresponding to the discrete and loop-free production rules from the hybrid program grammar above. Vectors of variables are denoted as $\bar{x}$ while x is an individual variable.

Safety guarantees in $d\mathcal{L}$ typically have the form:

$$\text{safety}(\text{pre}, \text{ctl}, \text{env}, \text{post}) \equiv \text{pre} \rightarrow [(\text{ctl}; \text{env})^*] \text{post}, \quad (1)$$

where pre defines initial conditions and post the safety guarantee; ctl and env describe the (nondeterministic) *control envelope* and the environment, respectively.

2) *Differential Game Logic*: Differential Game Logic ($d\mathcal{GL}$) extends $d\mathcal{L}$ to reason about *hybrid games*, i.e., hybrid, noncooperative zero-sum games between two players, *Angel* and *Demon*. A game is expressed as a program α in a syntax extending the hybrid program grammar as summarized below, however, for a comprehensive introduction, see [47].

For a game α , $[\alpha] Q$ states that Demon has a winning strategy to achieve Q , while $\langle \alpha \rangle Q$ indicates that Angel has a winning strategy. In the $d\mathcal{L}$ primitives, we assume Angel controls *nondeterministic decisions* by default, with control explicitly transferred to Demon via additional primitive α^d .

In this interpretation, a $d\mathcal{L}$ safety property $P \rightarrow [\alpha] Q$ for a classic hybrid program (i.e., without duality) states that, when starting from P , Demon has a winning strategy to ensure Q regardless of Angel's nondeterministic decisions. For example, the game $\alpha \equiv (v := 1 \cup v := -1)^d; x' = v$ states that Demon can choose $v \in \{-1, 1\}$ (observe the duality operator) and subsequently Angel controls for how long to evolve x along the differential equation. The formula $x \neq 0 \rightarrow [\alpha] x \neq 0$ is then valid: Demon can choose v such that, even as Angel controls the evolution of x , x never reaches 0. Specifically, Demon selects $v := 1$ iff $x > 0$, ensuring $x \neq 0$. Hence, Demon has a winning strategy for $x \neq 0$. Although $d\mathcal{GL}$ has a substantially different semantical foundation [47], it aligns with $d\mathcal{L}$ semantics for programs and formulas without duality [53, p. 520ff].

3) *Verifying NN with VerSAILLE*: As indicated in Sec. III-A1, $d\mathcal{L}$ is used to prove the safety of abstract control envelopes, typically formulated as in Formula (1). For real-world systems, however, we must relate these abstract safety proofs to concrete controller implementations outside $d\mathcal{L}$. This requires showing that the idealized implementation *refines* the safe control envelope [54], [55], i.e., that *all* behaviors of the idealized implementation are also possible in the $d\mathcal{L}$ model [54], [55]. To this end, ModelPlex [44] can derive correct-by-construction *controller monitor formulas*. Given a control envelope ctl , its monitor formula χ_{ctl} is a first-order

formula over pre- and post-states, satisfied only if a state transition is modeled by ctl . Formally:

Definition 1 (Correct Controller Monitor [44]). A controller monitor formula χ_{ctl} is correct for a hybrid program controller ctl with bound variables $x_1, \dots, x_n$ iff the following $d\mathcal{L}$ formula is valid: $\chi_{\text{ctl}} \rightarrow \langle \text{ctl} \rangle \bigwedge_{i=1}^n x_i = x_i^+$.

At least, χ_{ctl} typically includes the variables $x_1, x_1^+, \dots, x_n, x_n^+$ describing the bound variables of ctl before and after the execution of ctl .

Originally used for constructing monitors and shields [1], [45], ModelPlex was recently shown to support a priori safety verification through VerSAILLE [32]. Given a safe $d\mathcal{L}$ model (Formula (1)) and a loop invariant χ_{inv} , ModelPlex derives the controller monitor formula χ_{ctl} . We model the NN's behavior as a before-after predicate $\text{impl}(\bar{x}, \bar{x}^+)$. Verifying infinite-horizon safety of the NNCS then reduces to proving the following real arithmetic formula [32, Thm. 1]:

$$\chi_{\text{inv}} \wedge \text{impl}(\bar{x}, \bar{x}^+) \rightarrow \chi_{\text{inv}} \quad (2)$$

The obtained guarantee has a rigorous foundation in $d\mathcal{L}$: Based on $\text{impl}(x, x^+)$, we derive a hybrid program (called *nondeterministic mirror*, denoted $\alpha_{\text{refl}}(\text{impl})$) mirroring the behavior of the before-after predicate *within* $d\mathcal{L}$. Proving Formula (2) then implies safety when replacing ctl with $\alpha_{\text{refl}}(\text{impl})$.

B. Efficient Code Generation

We review the background on fixed-point arithmetic and mixed-precision tuning here.

1) *Fixed-Point Arithmetic*: Fixed-point numbers represent values as integers with an (implicit) format $\langle s, Q, \pi \rangle$, where $s \in \{0, 1\}$ indicates the presence of a sign bit, $Q \in \mathbb{N}$ is the total word length, and $\pi \in \mathbb{N}$ specifies the binary point position (counted from the least significant bit). Bit allocation defines an integer part, I , determining the range $[-2^I, 2^I]$, and a fractional part, π , controlling the *precision*: more fractional bits yield higher precision. Assuming truncation as the rounding mode (default for synthesis tools like Xilinx Vivado [42]), the maximum roundoff error is $2^{-\pi}$. Fixed-point operations use efficient integer arithmetic and bit-shifting [56], making them ideal for resource-constrained systems.

2) *Mixed-Precision Tuning*: Mixed-precision tuning assigns different bitwidths to variables and constants to optimize resource usage while satisfying error bounds. Several techniques [41], [57]–[59] have been proposed to automate this process. In this work, we focus on the tool Daisy [41] and briefly summarize its approach below.

Daisy [41] performs mixed-precision tuning by heuristically navigating the search space using *delta debugging* [60]. To soundly compute roundoff errors during the search, Daisy applies either interval or affine arithmetic, depending on the configuration. The process starts from a minimal bit length, uniformly increasing the precision to meet the error target, and then recursively partitioning variables to selectively lower bitwidths. Among multiple feasible assignments, Daisy selects the one minimizing an *area-based* cost function. The resulting

mixed-precision fixed-point code can be compiled by synthesis tools like the Xilinx HLS compiler. Daisy is most effective for straight-line numerical code without complex control structures such as loops, conditionals, or arrays.

IV. SAFETY UNDER PERTURBATION

As discussed in Sec. III-A, VerSAILLE’s safety guarantees rely on idealized NN implementations in real arithmetic. This has the advantage of enabling powerful real-arithmetic solvers [61]–[64] and real-valued NN verification tools (as reported in recent surveys and competitions [65]–[67]). However, it raises the question of whether these safety results hold when the NN is implemented in fixed-point arithmetic. In practice, real-world systems deviate from their idealized real-valued models along three dimensions:

- (I1) *Sensor Readouts*: Limited bandwidth or finite measurement accuracy affects input precision.
- (I2) *Roundoff Errors*: Errors can arise during computations.
- (I3) *Actuator Limitations*: Physical constraints may impact the precision of chosen control actions.

In principle, such factors can be mitigated by designing control envelopes for environments that explicitly model sensor and actuator noises, and computation errors. For instance, the hybrid program $\alpha_{\text{noise}} \equiv (\varepsilon_v := *; ?(|\varepsilon_v| \leq \delta_v); v := v + \varepsilon_v)$ models bounded perturbations in computing v in the robot example from Sec. IV-A. However, many case studies, e.g., [32], [68]–[70], only assume ideal sensing, actuation, and computation for simplicity. This is unsurprising as hybrid system analysis is already complex (even undecidable [71]) even without accounting for finite-precision effects.

This work proposes a technique to check control envelope robustness to sensing, actuation, and roundoff errors, given a safe control envelope oblivious to such perturbations. We further show how these control envelopes can be used to verify fixed-point implementations.

We begin by recalling how VerSAILLE guarantees safety for an idealized implementation for the robot example (Sec. IV-A). Building on this, our contribution starts with the formalization of perturbations (Sec. IV-B), a robustness property for control envelopes and decidable conditions for proving it (Sec. IV-C). We then extend VerSAILLE to support verification under perturbation (Sec. IV-D). While our method is illustrated using the running example — with some formal notions deferred to the extended version — Sec. V shows its applicability across realistic case studies.

A. Setup: Idealized Implementation Safety with VerSAILLE

We recall how $d\mathcal{L}$ and VerSAILLE can derive safety guarantees for an idealized implementation of our robot example from Figure 2. The system state is given by position p and velocity v along with a clock variable t tracking elapsed time. We formalize the robot’s physical behavior ($p' = v$), or *plant model*, as a hybrid program env ensuring control actions occur at least every T seconds (see details in extended version).

Different control scenarios and corresponding pre/postconditions and control envelopes are shown in

$$\begin{aligned} \text{ctl}_1^1 &\equiv (v := *; ?(0 \leq v \leq V_{\max} \wedge Tv \leq p)) \\ \text{ctl}_1^2 &\equiv (v := *; ?(-V_{\max} \leq v \leq V_{\max} \wedge 0 \leq p - Tv)) \\ \text{ctl}_2^2 &\equiv (v := *; ?(-V_{\max} \leq v \leq V_{\max} \wedge 0 \leq p - Tv \leq W)) \\ \text{pre}_1 &\equiv p, V_{\max} \geq 0 \wedge T > 0 \quad \text{post}_1 \equiv 0 \leq p \\ \text{pre}_2 &\equiv \text{pre}_1 \wedge p \leq W > 0 \quad \text{post}_2 \equiv 0 \leq p \leq W \end{aligned}$$

Fig. 3. Possible control envelopes, pre- and postconditions for robot (Sec. II)

Figure 3. In the single-wall case, the safety property post_1 is guaranteed by the envelopes ctl_1^1 and ctl_1^2 , which differ in whether bidirectional motion is allowed. For the two-wall case, the robot must stay within bounds defined by post_2 , requiring a different control envelope ctl_2^2 .

With suitable loop invariants, we can prove safety for ctl_1^1 and ctl_1^2 w.r.t. pre_1 and post_1 , as well as safety for ctl_2^2 w.r.t. pre_2 and post_2 in the theorem prover KeYmaera X [72] (proof in extended version). The envelope ctl_1^1 slightly deviates from the controller in [45], which checks for $Tv \leq p$ (instead of $Tv \leq p$). This change makes the envelope more *permissive* by allowing intermediate velocities $0 < v < V_{\max}$ when choosing $v = V_{\max}$ would no longer be safe; however, ctl_1^1 ’s behavior directly depends on the relation between the (changing) variables p , v and the computed Tv .

We can now use the $d\mathcal{L}$ safety results to verify an idealized NN implementations. For demonstration, we assume the idealized implementation is described by the predicate $\text{impl}_{\mathbb{R}}(p, v^+) \equiv (v^+ = \frac{-1}{0.01(p+10)} + M)$. Infinite-time horizon safety for one-wall, unidirectional case then be verified by instantiating Formula (2) and proving the following:

$$\underbrace{p \geq 0 \wedge \text{impl}_{\mathbb{R}}(p, v^+)}_{\chi_{\text{inv}}^{1,1}} \rightarrow \underbrace{(0 \leq v^+ \leq V_{\max} \wedge 0 \leq p - Tv^+)}_{\chi_{\text{ctrl}}^{1,1}}$$

where $\chi_{\text{inv}}^{1,1}$ is the loop invariant and $\chi_{\text{ctrl}}^{1,1}$ is the controller monitor derived from ctl_1^1 via ModelPlex. In practice, this validity is not proven directly by Formula (2), but via highly tuned NN verification tools that implicitly reason about impl .

For $M = V_{\max} = 10$, we prove validity of the formula using an SMT solver. Let $\alpha_{\text{refl}}(\text{impl}_{\mathbb{R}})$ be the non-deterministic mirror of $\text{impl}_{\mathbb{R}}$, i.e., the hybrid program capturing its behavior. This implies the safety predicate $\text{safety}(\text{pre}_1, \alpha_{\text{refl}}(\text{impl}_{\mathbb{R}}), \text{env}, \text{post}_1)$. Using the same approach for ctl_2^2 , we also prove safety for $\text{impl}_{\mathbb{R}}(p, v^+)$ for $M = V_{\max} = 10, W = 100$. However, VerSAILLE’s analysis assumes real arithmetic. Hence, it ignores roundoff errors introduced by realistic, fixed-point implementations of $\text{impl}_{\mathbb{R}}$.

B. Formalizing Perturbations

We formalize perturbations as concrete, discrete, loop-free hybrid programs. As seen earlier, α_{noise} captures *post*-controller perturbations to v , which may represent roundoff errors, actuation errors, or both (with δ_v as maximal roundoff, maximal deviation in actuation, or the combined bound). However, sensor perturbations require modifying the state *before* controller execution. To cover all perturbation profiles (I1–I3), we formalize general perturbation as a tuple $(\alpha_{\text{pre}}, \alpha_{\text{post}})$,

where α_{pre} applies before the controller and α_{post} after. We call such pairs *angelic perturbations*:

Definition 2 (Angelic Perturbation). *An angelic perturbation consists of two discrete, concrete, loop-free hybrid programs $\alpha_{\text{pre}}, \alpha_{\text{post}}$. We denote an angelic perturbation as a tuple (i.e. $(\alpha_{\text{pre}}, \alpha_{\text{post}})$).*

In the robot example, we consider two such perturbation profiles:

$$\begin{aligned} \text{angel}_{\text{pre}}^1 &\equiv ?(\top) \quad \text{angel}_{\text{post}}^1 \equiv \alpha_{\text{noise}} \\ \text{angel}_{\text{pre}}^2 &\equiv p^- := p; \varepsilon_p := *; ?(|\varepsilon_p| \leq \delta_p); p := p + \varepsilon_p \\ \text{angel}_{\text{post}}^2 &\equiv p := p^-; \varepsilon_v := *; ?(|\varepsilon_v| \leq \delta_v); v := v + \varepsilon_v \end{aligned}$$

Here, $(\text{angel}_{\text{pre}}^1, \text{angel}_{\text{post}}^1)$ perturbs only v by at max δ_v ; whereas, $(\text{angel}_{\text{pre}}^2, \text{angel}_{\text{post}}^2)$ perturbs both the sensor input p without affecting the system's dynamics (the original value is restored after control execution) and the control output v .

C. Envelope Robustness

Before verifying implementations, we must check if the envelope itself is robust to angelic perturbations—a necessary condition for subsequent verification as we use the envelope to check if our implementations represent a winning strategy.

We formulate the robustness of control envelopes as a game: rather than requiring that *all* control actions remain safe under perturbation, we require that for any state, *there exists* a control action that 1) is permitted according to the envelope, and 2) guarantees safety even when perturbed by the given angelic perturbation. The *good* Demon, choosing an action from within the envelope, hence must take into account the perturbations of the *bad* Angel.

To check this property, we formulate a $d\mathcal{GL}$ game where Demon chooses from ctl and Angel controls perturbations and the environment. Robustness (in $d\mathcal{GL}$) is then defined as Demon having a winning strategy to guarantee post:

Definition 3 (Robustness). *A control envelope ctl with a safety guarantee $\text{safety}(\text{pre}, \text{ctl}, \text{env}, \text{post})$ is robust to angelic perturbations $(\text{angel}_{\text{pre}}, \text{angel}_{\text{post}})$ iff the following is valid:*

$$\text{pre} \rightarrow \left[\left(\text{angel}_{\text{pre}}; (\text{ctl})^d; \text{angel}_{\text{post}}; \text{env} \right)^* \right] \text{post} \quad (3)$$

Definition 3 assumes that ctl already has a real-valued safety guarantee (e.g. from [32], [68]–[70]). We *reuse* this knowledge (and the safety proof) to check the robustness of ctl utilizing insights on the decidability of differential refinements [55]. We propose a practical, *sufficient* criterion for controller robustness that can be checked in decidable real arithmetic (see extended version for proofs and additional simplifications).

Theorem 1 (Robustness of Control Envelopes). *Let ctl be as in Definition 3 and χ_{inv} be the inductive invariant used to prove $\text{safety}(\dots)$. For angelic perturbation $\text{angel}_{\text{pre}}, \text{angel}_{\text{post}}$ and monitor formulas $\chi_{\text{ctrl}}, \chi_{\text{pre}}, \chi_{\text{post}}$ for*

$\text{ctl}, \text{angel}_{\text{pre}}, \text{angel}_{\text{post}}$ the validity of the following real arithmetic formula implies the validity of Equation (3).

$$\forall \bar{x}_0 \forall x_1 \exists \bar{x}_2 \forall \bar{x}_3 \left(\chi_{\text{inv}}(\bar{x}_0) \wedge \chi_{\text{pre}}(\bar{x}_0, \bar{x}_1) \rightarrow \left(\chi_{\text{ctrl}}(\bar{x}_1, \bar{x}_2) \wedge (\chi_{\text{post}}(\bar{x}_2, \bar{x}_3) \rightarrow \chi_{\text{ctrl}}(\bar{x}_0, \bar{x}_3)) \right) \right) \quad (4)$$

Liveness: When both $\text{angel}_{\text{pre}}$ and $\text{angel}_{\text{post}}$ are set to the skip program $?(\top)$, the robustness property reduces to a standard *liveness* property, i.e., whether the control envelope (without perturbation) contains a safe action in *all* reachable states. In this sense, robustness can be seen as liveness under input/output perturbations.

Robot Example: As noted in Sec. II, the robot restricted to moving toward the wall may crash if velocity computation is even slightly perturbed. Using the formalization from Sec. IV-A and our robustness notion, we can state that the ctl_1^1 is not robust w.r.t. the angelic perturbation $(\text{angel}_{\text{pre}}^1, \text{angel}_{\text{post}}^1)$, as confirmed by a counterexample to the corresponding $d\mathcal{GL}$ formula in Definition 3 in KeYmaera X (see also extended version's appendix). In contrast, for $\delta_v \leq V_{\text{max}}/2$, we prove via Theorem 1 that ctl_1^2 is robust w.r.t. angelic perturbation $(\text{angel}_{\text{pre}}^1, \text{angel}_{\text{post}}^1)$. Similarly, ctl_2^2 is robust under the additional assumption that $\delta_v * T \leq W/2$, ensuring sufficient distance between the *two* walls for safe operation under worst-case velocity perturbations.

D. Implementation Safety under Perturbation

We have shown how to check the robustness of control envelopes by proving that each state admits *one* safe action. However, this is insufficient for concrete implementations, whether idealized or fixed-point, where *every* chosen action must ensure safety.

In $d\mathcal{L}$ terms, given a before-after predicate for an implementation impl and its $d\mathcal{L}$ representation $\alpha_{\text{refl}}(\text{impl})$, safety under perturbation requires proving:

$$\text{pre} \rightarrow \left[\left(\text{angel}_{\text{pre}}; \alpha_{\text{refl}}(\text{impl}); \text{angel}_{\text{post}}; \text{env} \right)^* \right] \text{post} \quad (5)$$

By translating angelic perturbations into real arithmetic [44], [55], we derive a decidable, real arithmetic condition for safety under perturbation (see proof in extended version).

Theorem 2 (Safety under Perturbation). *Let ctl be like in Definition 3 and χ_{inv} be the inductive invariant used to prove $\text{safety}(\dots)$. For angelic perturbation $\text{angel}_{\text{pre}}, \text{angel}_{\text{post}}$ and monitor formulas $\chi_{\text{ctrl}}, \chi_{\text{pre}}, \chi_{\text{post}}$ for $\text{ctl}, \text{angel}_{\text{pre}}, \text{angel}_{\text{post}}$ the validity of the formula below implies Formula (5).*

$$\forall \bar{x}_0 \forall x_1 \forall \bar{x}_2 \forall \bar{x}_3 \left(\chi_{\text{inv}}(\bar{x}_0) \wedge \chi_{\text{pre}}(\bar{x}_0, \bar{x}_1) \wedge \text{impl}(\bar{x}_1, \bar{x}_2) \wedge \chi_{\text{post}}(\bar{x}_2, \bar{x}_3) \rightarrow \chi_{\text{ctrl}}(\bar{x}_0, \bar{x}_3) \right) \quad (6)$$

Phrased differently, proving Formula (6) shows that impl encodes a *winning strategy* for demon in the game of Equation (3) (with ctl substituted by $\alpha_{\text{refl}}(\text{impl})$). For NNs, verification is typically done using NN verification tools, not general real arithmetic solvers. In this setting, Formula (6) (with impl omitted) serves directly as the NN's specification and can be simplified further (see extended version).

Robot Example: We previously showed that ctl_1^1 and ctl_1^2 are robust under angelic perturbation $(\text{angel}_{\text{pre}}^1, \text{angel}_{\text{post}}^1)$. We now examine whether the idealized implementation $\text{impl}_{\mathbb{R}}(p, v^+)$ (Sec. IV-A) is safe under perturbation.

To this end, we instantiate Formula (6) for all combinations of $\{\text{ctl}_1^1, \text{ctl}_1^2\} \times \{(\text{angel}_{\text{pre}}^1, \text{angel}_{\text{post}}^1), (\text{angel}_{\text{pre}}^2, \text{angel}_{\text{post}}^2)\}$ w.r.t. $\text{impl}_{\mathbb{R}}$, yielding four real arithmetic formulas. Each formula’s validity implies the safety of implementation $\text{impl}_{\mathbb{R}}$ w.r.t. its environment (one/two walls) and chosen perturbation.

For $T = 1.0$, $V_{\max} = 10$, $W = 100$, $M = 9.5$ and $\delta_p = \delta_v = 0.25$ all 4 formulas are proven using KeYmaera X. Hence, any fixed-point implementation differing from $\text{impl}_{\mathbb{R}}$ by at most 0.25 and with sensor perturbations of ≤ 0.25 remains safe within the fixed domain $p \in [0, 100]$. We now use the safety result in Sec. IV-E to generate safe, efficient fixed-point code.

Importantly, we previously disproved the robustness of ctl_1^1 w.r.t. $(\text{angel}_{\text{pre}}^1, \text{angel}_{\text{post}}^1)$. Hence, as robustness is a necessary condition, no matter the parameter choice, we cannot prove safety of $\text{impl}_{\mathbb{R}}$ w.r.t. ctl_1^1 under the given angelic perturbations. Without checking robustness, a failed safety proof under perturbation leaves unclear if this is due to a limitation of the fixed-point implementation or of the chosen $d\mathcal{L}$ model.

E. Implementation Synthesis

With $\text{impl}_{\mathbb{R}}(p, v^+)$ proven safe under bounded perturbations, we use state-of-the-art mixed-precision tuning to synthesize a fixed-point implementation that stays within the chosen bound δ_v . For this, we use Daisy [41] (also supports uniform precision) that performs tuning using interval or affine arithmetic. It generates C++ code with mixed-precision fixed-point arithmetic (up to 64-bit width), which can be directly compiled by Xilinx [42] to measure runtime in machine cycles.

However, Daisy does not natively support vectors, matrices, or loops, common in NNs. It requires unrolled structures and loops and also generates fully unrolled code, which may overwhelm Xilinx [39]. To address this, we add a pre-processing step to automatically unroll standard NN structures for Daisy and a post-processing step to reintroduce the structures and loops after tuning. This enables Daisy to handle our neural networks effectively. Note that we also considered the tool Aster [39], which supports NN-specific code. But it is limited to 32-bit precision due to solver constraints and becomes infeasible for deeper networks due to imprecise range overapproximations, making it unsuitable for our purpose.

For our robot example, we first used extended Daisy to generate a 32-bit uniform fixed-point implementation of $\text{impl}_{\mathbb{R}}(p, v^+)$, yielding a worst-case roundoff error of 5.37×10^{-6} and a latency of 68 cycles. Since we proved safety under perturbations up to $\delta_v = 0.25$, Daisy can synthesize a more efficient mixed-precision implementation using at most 17 bits, reducing latency to 30 cycles while still ensuring safety.

F. Limitations

As stated in Theorem 1, Formula (4) implies controller robustness. However, if Formula (4) is not valid, this does not

necessarily mean the controller is non-robust. A counterexample $\bar{x}_0$ satisfying pre is a concrete violation, but one outside of pre may be spurious; robustness might still hold under a *stronger* loop invariant that excludes $\bar{x}_0$.

Thus, our control envelope approach is, in general, incomplete. However, many $d\mathcal{L}$ proofs provide guarantees for the weakest assumptions and thus set the invariant region as the initial condition proving safety results for very permissive control envelopes: If $\text{pre} \leftrightarrow \chi_{\text{inv}}$ (and not only $\text{pre} \rightarrow \chi_{\text{inv}}$), then any counterexample is a concrete violation w.r.t. actions deemed safe by ctl .

Finally, our method relies on proven, safe $d\mathcal{L}$ control envelopes. Fortunately, a wide range of envelopes exists across applications [68]–[70], [73]–[76], and recent work [77] shows that their synthesis can be automated.

V. EVALUATION

We evaluate our methodology on three realistic case studies to demonstrate its practical applicability. We begin with Adaptive Cruise Control (ACC), verifying safety for both discrete and continuous action spaces. We then apply our approach to the more complex and safety-critical Vertical Airborne Collision Avoidance System (VCAS), specifically focusing on two neural networks—Do-Not-Climb (DNC) and Do-Not-Descend (DND)—which issue collision avoidance advisories in case the current advisory is DNC/DND.

While the NNs for continuous ACC and VCAS are taken from prior work [4], [5], [32], the discrete ACC networks were trained specifically for this paper. The continuous ACC network has 256 ReLU nodes and over 12k parameters; the discrete variant has the same architecture but has 3 output nodes instead of 1. Each of the two networks used in the VCAS case study contains 6x45 ReLU nodes. Table I summarizes our experimental results.

A. Experimental Setup

All experiments were conducted on an Ubuntu 22.04 system with an AMD Ryzen 7 CPU and 32 GB RAM. Safety proofs in differential dynamic logic and real arithmetic were performed interactively using KeYmaeraX [72], which internally relies on Mathematica [64] for quantifier elimination. NN verification was done using N^3V [32], [78], which supports polynomial specifications with arbitrary propositional structure. Internally, N^3V is based on Z3 [61], [62], PicoSAT [79] and the linear specification NN verifier nenum [16]. For mixed-precision code generation and FPGA synthesis, we used Daisy’s March 2, 2021 version (with no major updates since then) and Xilinx’s Vitis HLS [42] (version 2023.1), downloaded on September 24, 2024.

B. Case Study 1: Continuous Adaptive Cruise Control

For continuous ACC, prior work [32] provides a $d\mathcal{L}$ safety guarantee for NN verification. In this setting, an *ego car* (following a *front car*) chooses a relative acceleration a_{rel} based on current relative position p_{rel} and velocity v_{rel} . The safety goal is to prevent collisions while ensuring the ego car

TABLE I

SUMMARY OF RESULTS. δ : WORST-CASE PERTURBATION/ERROR BOUND; TIME ($\mathbb{R}$): VERIFICATION TIME WITH REALS; TIME (δ): VERIFICATION TIME UNDER PERTURBATION; NN SYNTHESIS TIME: TIME TO GENERATE AN NN WITH ERROR BOUND δ ; CYCLES: LATENCY REPORTED BY XILINX.

Case Study	δ	NN Verification		NN Synthesis	
		Time ($\mathbb{R}$)	Time (δ)	Time	# Cycles
ACC (cont.)	1.0	2.02m	9.3m	1.53h	567
ACC (disc.)	0.01	59s	1m	2.01h	579
VCAS DNC	25^{-3}	18.92m	27.45m	1.73h	655
DND	25^{-3}	15.75m	19.1m	1.82h	656

does not fall behind. Previous work had also established a real-valued safety guarantee for a validated NN [32, Sec. 5], raising the question whether the same safety guarantee holds in a finite precision setting. To this end, we formalize an angelic perturbation as $(?(\top), \text{angel}_{\text{post}}^{\text{acc}})$ with:

$$\text{angel}_{\text{post}}^{\text{acc}} \equiv (\varepsilon_{a_{\text{rel}}} := *; ?(|\varepsilon_{a_{\text{rel}}}| \leq \delta_{a_{\text{rel}}}); a_{\text{rel}} := a_{\text{rel}} + \varepsilon_{a_{\text{rel}}})$$

Envelope Robustness: Initial analysis showed that the control envelope derived before is not robust under $(?(\top), \text{angel}_{\text{post}}^{\text{acc}})$. It assumes actions in the range $a_{\text{rel}} \in [-B, A]$, where $-B$ is the maximum braking acceleration, and guarantees safety as long as braking with $-B$ avoids a crash. However, this relies on the *precise* execution of $-B$, which is unrealistic under finite precision. We therefore revised the control envelope to reflect this limitation and formally proved its safety and robustness in KeYmaera X.

NN Verification: Using the revised control envelope, we derived the specification for the NN as described in Sec. IV-E and verified the NN using N^3V . Table I reports the verification times for both the original (Time $\mathbb{R}$) and perturbation-aware (Time δ) specifications. While verification under perturbation takes longer, it remains feasible. The longer time is likely due to the increased propositional complexity of the specification, leading to more queries to the underlying verifier nenum.

Code Generation: Finally, we used Daisy [41] to generate an efficient mixed-precision fixed-point C++ implementation of the NN that meets the error bound $\delta = 1.0$. To compute latency, we compiled the generated code with Xilinx Vivado HLS [42] for a standard FPGA architecture. Note that this latency is exact, i.e., we do not consider noisy runtime measurements from actual hardware.

With our pre- and post-processing pipeline, Daisy successfully generated code with an error of 1.0. While uniform precision required 32 bits, mixed-precision tuning produced an implementation using a mix of 31- and 32-bit variables, reducing total bit usage by 3.66% compared to the uniform version. The latency remained at 567 cycles, as the limited reduction in bit widths was not sufficient to impact the overall execution time. We note that retaining the fully unrolled version could potentially reduce latency, but the size may overwhelm Xilinx.

Discussion: The continuous ACC case study illustrates that the proposed approach presented in this paper applies effectively to realistic regression-based NN control systems, handling roundoff errors.

While NN verification typically scales with the number of ReLU nodes, mixed precision tuning and code generation scale with the number of parameters of the NN, *including* all weights and biases. Since both steps are performed only once, our method remains practical even for networks with 256 ReLU nodes and over 12K parameters.

Most importantly, the proposed approach enhances *warranted* trust in NN controllers by ensuring *infinite-time horizon* safety and bridging the gap between theoretical guarantees in real arithmetic and their efficient, deployable code on FPGAs.

C. Case Study 2: Discrete Adaptive Cruise Control

In this case study, the NNs are classifiers selecting one of three actions ($a_{\text{rel}} \in -B, 0, A$) via an argmax over three outputs. Finite-precision analysis is more challenging here, as *any* error can flip decisions for *some* inputs near classification boundaries. For instance, if $a_{\text{rel}} = -B$ is selected for inputs x such that $\langle w, y \rangle \leq b$, then for input values with $\langle w, y \rangle = b$, even infinitesimal perturbations can flip the action.

Hence, in this case study, the key question is not *if* the NN’s behavior changes under perturbations but *where* it changes. For the chosen control envelope and idealized implementation we verified safety under perturbation for a trained NN for $\delta = 0.01$ which simultaneously shows the envelope’s robustness.

With a perturbation bound of 0.01, Daisy estimated that 32-bit uniform precision was sufficient to meet the error requirements. We then used Daisy to synthesize a mixed-precision implementation, assigning 29 to 32 bits across variables. The mixed-precision tuning took 2.01 hours and produced code with a latency of 579 cycles, achieving a 0.69% reduction in latency and a 2.14% improvement in total bit usage compared to the 32-bit uniform precision implementation.

D. Case Study 3: Vertical Airborne Collision Avoidance

Our third case study focuses on the Vertical Airborne Collision Avoidance System (VCAS), developed through multiple Federal Aviation Administration (FAA)-supervised research efforts [80], [81]. It aims to prevent Near Mid-Air Collisions (NMACs), defined as aircraft coming within 500 feet horizontally or 100 feet vertically, by identifying dangerous trajectories and issuing timely advisories to adjust flight paths.

Given that hazardous situations often develop rapidly, the advisory system must provide *correct* decisions. Previous work has formalized this CPS in $d\mathcal{L}$ [68] and proposed NN controllers for use [4], [5]. Although the full combination of these NNs is unsafe, two specific NNs (DNC and DND) were proven safe under idealized real-valued semantics [32]. Like in the discrete ACC, these controllers work in a discrete action space with similar challenges under finite precision.

VCAS represents a critical application where providing formal safety guarantees for fixed-point implementations—not just idealized models—is crucial. To this end, we applied our methodology to the two verified NNs, constructing an angelic perturbation model analogous to $\text{angel}_{\text{post}}^{\text{acc}}$ to reflect realistic numerical and actuation uncertainties. We then derived the corresponding NN safety specifications.

Using N^3V , we verified both NNs under this new specification in comparable time to the original setting. Daisy’s analysis showed that 42-bit uniform precision is required to meet the perturbation bound of 25^{-3} . We then synthesized mixed-precision implementations in 1.73 hours (DNC) and 1.82 hours (DND). While the latencies remained the same at 655 and 656 cycles w.r.t. the uniform baselines, mixed-precision versions have bit savings of 6.18% and 5.2%, respectively.

E. Discussion

Our three case studies demonstrate that our approach scales to safety-critical, real-world systems like VCAS and applies to both regression and classification-based NN Control Systems. Importantly, our methodology provides *global* safety guarantees for the NN, which are generally more challenging to obtain and scale less than local robustness guarantees. However, many control applications of interest rely on NNs of comparable size [3], making our approach practically relevant.

VI. RELATED WORK

Safety Analysis: While many techniques guarantee safety for Cyber-Physical Systems (CPS) control, most of them omit finite-precision or sensor/actuation errors. In principle, $d\mathcal{L}$ [46], [48], [50] supports reasoning about such errors; however, case studies often avoid it due to complexity [68]–[70]. Our approach addresses this gap by enabling robustness analysis of control envelopes from such studies.

VerSAILLE [32] is the only method using $d\mathcal{L}$ guarantees for NNCSs. This enables *infinite-time* horizon guarantees. In contrast, closed-loop reachability [20]–[25], [27]–[31], [82] ensures safety only over a finite horizon, limiting inductive reasoning [32, Appendix E.2]. Similarly, barrier certificates [83]–[87] lack the exact reasoning supported by VerSAILLE, and typically cover smaller regions. Unlike our work, all mentioned NNCS verification works *ignore* the effects of fixed-point arithmetic which are essential for *realistic* safety guarantees. Unlike [40], which integrates closed-loop reachability with bounding fixed-point errors, we extend VerSAILLE and can thus provide stronger, infinite-time horizon guarantees.

Recent work [88] analyzes divergence between closed-loop NN controllers using ε -equivalence [89], [90] and Lipschitz analysis, but focuses on finite-horizon settings and does not address sound quantization. For a broader comparison between closed-loop verification, barrier certificates, and VerSAILLE, we refer to [32, Sec. 5, 6, Appendix E.2].

Efforts to verify quantized NNs [38], [91], [92] use bit-vector constraints. These tools are incompatible with the real-valued polynomial constraints required by VerSAILLE, limiting their use for infinite-horizon NNCS verification.

Mixed Precision Tuning: For sound mixed-precision tuning, Daisy [41] (used in this work) and POPiX [59] support fixed-point arithmetic, while FPTuner [57], Salsa [58], and POP [93] focus on floating-point programs. Unlike Daisy, POPiX frames tuning as an ILP problem but relies on dynamic range analysis, which compromises soundness. Daisy also integrates expression rewriting to improve mixed-precision

tuning, though this added step increases computational cost, making it harder to scale for larger benchmarks.

Neural Network Quantization: State-of-the-art NN quantization [94]–[97] typically target NN classifiers, optimizing accuracy without formal guarantees. Recent efforts, such as Aster [39], automate mixed-precision tuning for NN controllers under error bounds but suffer from imprecise integer range estimation, limiting their usability for *deeper* networks. Popinns [98] also synthesizes bounded fixed-point implementations but relies on dynamic analysis and assumes access to a floating-point model, unlike our approach, where the goal is sound efficient code generation.

Combined Approaches: VeriPhy [45] synthesizes sandboxes based on $d\mathcal{L}$ contracts and interval-based fallback controllers. However, it provides guarantees around, not *of*, the controller itself. Extending VeriPhy to realistic NNs would require substantial changes, as NNs are not amenable to interactive proofs. We view VeriPhy as orthogonal to our approach, applicable when direct NN verification is impossible.

Finally, for code generation, we use Xilinx Vivado [42] for traditional and NN controllers. For neural controllers, tools like FloPoCo [99] can generate optimized FPGA-based dot products, which we leave as future work.

VII. CONCLUSION

This paper addresses the critical gap between theoretical infinite-horizon safety guarantees for NN-controlled systems and their practical finite-precision implementations. Since real-valued guarantees may fail under finite precision, we formulated control envelope robustness as a game between a *good* Demon and a *bad* Angel. Verifying infinite-horizon safety thus reduces to showing that the NN implements a winning strategy while keeping fixed-point implementation errors within allowable perturbations. Our approach relies on decidable real-arithmetic conditions, avoiding the need for undecidable $d\mathcal{GL}$ or $d\mathcal{L}$ validity proofs.

Our experiments on realistic NN control systems, from automotive to aeronautics, covering both regression- and classification-based NN architectures, demonstrate that accounting for perturbations introduces no significant overhead compared to perturbation-free verification, though costs may increase for more complex systems. While this paper focuses on NN controllers, the proposed technique naturally extends to other controller types. Overall, our work represents a significant step toward fostering *warranted* trust in real-world NN control systems.

REFERENCES

- [1] N. Fulton and A. Platzer, “Safe reinforcement learning via formal methods: Toward safe control through proof and learning,” in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, New Orleans, Louisiana, USA, February 2-7, 2018 (S. A. McIlraith and K. Q. Weinberger, eds.), pp. 6485–6492, AAAI Press, 2018.

- [2] M. Brosowsky, F. Keck, J. Ketterer, S. Isele, D. Slieter, and M. Zöllner, "Safe deep reinforcement learning for adaptive cruise control by imposing state-specific safe sets," in *2021 IEEE Intelligent Vehicles Symposium (IV)*, pp. 488–495, 2021.
- [3] T. T. Johnson, D. M. Lopez, L. Benet, M. Forets, S. Guadalupe, C. Schilling, R. Ivanov, T. J. Carpenter, J. Weimer, and I. Lee, "ARCH-COMP21 category report: Artificial intelligence and neural network control systems (AINNCS) for continuous and hybrid systems plants," in *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21)*, Brussels, Belgium, July 9, 2021 (G. Frehse and M. Althoff, eds.), vol. 80 of *EPiC Series in Computing*, pp. 90–119, EasyChair, 2021.
- [4] K. D. Julian and M. J. Kochenderfer, "Guaranteeing safety for neural network-based aircraft collision avoidance systems," *CoRR*, vol. abs/1912.07084, 2019.
- [5] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pp. 1–10, 2016.
- [6] H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel, "Efficient neural network robustness certification with general activation functions," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3–8, 2018, Montréal, Canada* (S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 4944–4953, 2018.
- [7] K. Xu, Z. Shi, H. Zhang, Y. Wang, K. Chang, M. Huang, B. Kailkhura, X. Lin, and C. Hsieh, "Automatic perturbation analysis for scalable certified robustness and beyond," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.
- [8] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C. Hsieh, "Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021*, 2021.
- [9] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C. Hsieh, and J. Z. Kolter, "Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6–14, 2021, virtual* (M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, eds.), pp. 29909–29921, 2021.
- [10] P. Henriksen and A. R. Lomuscio, "Efficient neural network verification via adaptive refinement and adversarial search," in *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August–8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020* (G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, eds.), vol. 325 of *Frontiers in Artificial Intelligence and Applications*, pp. 2513–2520, IOS Press, 2020.
- [11] P. Henriksen and A. Lomuscio, "DEEPSPLIT: an efficient splitting method for neural network verification via indirect effect analysis," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19–27 August 2021* (Z. Zhou, ed.), pp. 2549–2555, International Joint Conferences on Artificial Intelligence Organization, 2021.
- [12] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017, Proceedings, Part I* (R. Majumdar and V. Kuncak, eds.), vol. 10426 of *LNCS*, (Cham), pp. 97–117, Springer, 2017.
- [13] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett, "The Marabou framework for verification and analysis of deep neural networks," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15–18, 2019, Proceedings, Part I* (I. Dillig and S. Tasiran, eds.), vol. 11561 of *LNCS*, (Cham), pp. 443–452, Springer, 2019.
- [14] R. Bunel, J. Lu, I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar, "Branch and bound for piecewise linear neural network verification," *J. Mach. Learn. Res.*, vol. 21, pp. 42:1–42:39, 2020.
- [15] S. Bak, H. Tran, K. Hobbs, and T. T. Johnson, "Improved geometric path enumeration for verifying ReLU neural networks," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* (S. K. Lahiri and C. Wang, eds.), vol. 12224 of *LNCS*, (Cham), pp. 66–96, Springer, 2020.
- [16] S. Bak, "nenum: Verification of ReLU neural networks with optimized abstraction refinement," in *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings* (A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, eds.), vol. 12673 of *LNCS*, (Cham), pp. 19–36, Springer, 2021.
- [17] H. Zhang, S. Wang, K. Xu, L. Li, B. Li, S. Jana, C. Hsieh, and J. Z. Kolter, "General cutting planes for bound-propagation-based neural network verification," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), 2022.
- [18] Z. Shi, Q. Jin, Z. Kolter, S. Jana, C. Hsieh, and H. Zhang, "Neural network verification with branch-and-bound for general nonlinearities," *CoRR*, vol. abs/2405.21063, 2024.
- [19] H. Wu, O. Isac, A. Zeljic, T. Tagomori, M. L. Daggett, W. Kokke, I. Refaeli, G. Amir, K. Julian, S. Bassan, P. Huang, O. Lahav, M. Wu, M. Zhang, E. Komendantskaya, G. Katz, and C. W. Barrett, "Marabou 2.0: A versatile formal analyzer of neural networks," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II* (A. Gurfinkel and V. Ganesh, eds.), vol. 14682 of *LNCS*, (Cham), pp. 249–264, Springer, 2024.
- [20] S. Bogomolov, M. Forets, G. Frehse, K. Potomkin, and C. Schilling, "JuliaReach: a toolbox for set-based reachability," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16–18, 2019* (N. Ozay and P. Prabhakar, eds.), (New York, NY, USA), pp. 39–44, ACM, 2019.
- [21] C. Schilling, M. Forets, and S. Guadalupe, "Verification of neural-network control systems by integrating Taylor models and zonotopes," in *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelfth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pp. 8169–8177, AAAI Press, 2022.
- [22] H. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson, "Star-based reachability analysis of deep neural networks," in *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings* (M. H. ter Beek, A. McIver, and J. N. Oliveira, eds.), vol. 11800 of *LNCS*, (Cham), pp. 670–686, Springer, 2019.
- [23] H. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* (S. K. Lahiri and C. Wang, eds.), vol. 12224 of *LNCS*, (Cham), pp. 3–17, Springer, 2020.
- [24] W. Xiang, H. Tran, X. Yang, and T. T. Johnson, "Reachable set estimation for neural network control systems: A simulation-guided approach," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 32, no. 5, pp. 1821–1830, 2021.
- [25] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verifying the safety of autonomous systems with neural network controllers," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 1, pp. 7:1–7:26, 2021.
- [26] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig 2.0: Verification of neural network controllers using Taylor model preconditioning," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I* (A. Silva and K. R. M. Leino, eds.), vol. 12759 of *LNCS*, (Cham), pp. 249–262, Springer, 2021.
- [27] C. Huang, J. Fan, W. Li, X. Chen, and Q. Zhu, "ReachNN: Reachability analysis of neural-network controlled systems," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, pp. 106:1–106:22, 2019.
- [28] J. Fan, C. Huang, X. Chen, W. Li, and Q. Zhu, "ReachNN*: A tool for reachability analysis of neural-network controlled systems," in

Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings (D. V. Hung and O. Sokolsky, eds.), vol. 12302 of LNCS, (Cham), pp. 537–542, Springer, 2020.

- [29] S. Dutta, X. Chen, and S. Sankaranarayanan, “Reachability analysis for neural feedback systems using regressive polynomial rule inference,” in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019* (N. Ozay and P. Prabhakar, eds.), (New York, NY, USA), pp. 157–168, ACM, 2019.
- [30] M. E. Akintunde, E. Botoeva, P. Kouvaros, and A. Lomuscio, “Formal verification of neural agents in non-deterministic environments,” *Auton. Agents Multi Agent Syst.*, vol. 36, no. 1, p. 6, 2022.
- [31] 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*, vol. 23, no. 117, pp. 1–45, 2022.
- [32] S. Teuber, S. Mitsch, and A. Platzer, “Provably safe neural network controllers via differential dynamic logic,” in *Advances in Neural Information Processing Systems* (A. Globerson, L. Mackey, A. Fan, C. Zhang, D. Belgrave, J. Tomczak, and U. Paquet, eds.), Curran Associates, Inc., 2024.
- [33] A. Szász, B. Bánhelyi, and M. Jelasity, “No soundness in the real world: On the challenges of the verification of deployed neural networks,” in *Forty-second International Conference on Machine Learning*, 2025.
- [34] D. Lohar, E. Darulova, S. Putot, and E. Goubault, “Discrete choice in the presence of numerical uncertainties,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2381–2392, 2018.
- [35] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, “Compiling kb-sized machine learning models to tiny iot devices,” 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.), (New York, NY, USA), pp. 79–95, ACM, 2019.
- [36] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015* (F. R. Bach and D. M. Blei, eds.), vol. 37 of *JMLR Workshop and Conference Proceedings*, pp. 1737–1746, JMLR.org, 2015.
- [37] L. C. Cordeiro, M. L. Daggitt, J. Girard-Satabin, O. Isac, T. T. Johnson, G. Katz, E. Komendantskaya, A. Lemesle, E. Manino, A. Sinkarovs, and H. Wu, “Neural network verification is a programming language challenge,” in *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part 1* (V. Vafeiadis, ed.), vol. 15694 of LNCS, pp. 206–235, Springer, 2025.
- [38] T. A. Henzinger, M. Lechner, and D. Zikelic, “Scalable verification of quantized neural networks,” in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 3787–3795, AAAI Press, 2021.
- [39] D. Lohar, C. Jeangoudoux, A. Volkova, and E. Darulova, “Sound mixed fixed-point quantization of neural networks,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 5s, pp. 136:1–136:26, 2023.
- [40] H. Thevendhriya, S. Ghosh, and D. Lohar, “Towards precision-aware safe neural-controlled cyber-physical systems,” *IEEE Embedded Systems Letters (ESL)*, 2024.
- [41] E. Darulova, E. Horn, and S. Sharma, “Sound mixed-precision optimization with rewriting,” in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018* (C. Gill, B. Sinopoli, X. Liu, and P. Tabuada, eds.), pp. 208–219, IEEE Computer Society / ACM, 2018.
- [42] Vivado Lab Solutions, “Vivado design suite,” 2021.
- [43] S. Teuber, D. Lohar, and B. Beckert, “Of good demons and bad angels: Guaranteeing safe control under finite precision,” *CoRR*, vol. abs/2507.22760, 2025.
- [44] S. Mitsch and A. Platzer, “Modelplex: verified runtime validation of verified cyber-physical system models,” *Formal Methods Syst. Des.*, vol. 49, no. 1-2, pp. 33–74, 2016.
- [45] R. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer, “Veriphy: verified controller executables from verified cyber-physical system models,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018* (J. S. Foster and D. Grossman, eds.), (New York, NY, USA), pp. 617–630, ACM, 2018.
- [46] A. Platzer, “Differential dynamic logic for hybrid systems,” *J. Autom. Reason.*, vol. 41, no. 2, pp. 143–189, 2008.
- [47] A. Platzer, “Differential game logic,” *ACM Trans. Comput. Log.*, vol. 17, no. 1, p. 1, 2015.
- [48] A. Platzer, “A complete uniform substitution calculus for differential dynamic logic,” *J. Autom. Reas.*, vol. 59, no. 2, pp. 219–265, 2017.
- [49] A. Platzer and Y. K. Tan, “Differential equation invariance axiomatization,” *J. ACM*, vol. 67, no. 1, pp. 6:1–6:66, 2020.
- [50] A. Platzer, “The complete proof theory of hybrid systems,” in *Proceedings of the 2012 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012*, pp. 541–550, {IEEE} Computer Society, 2012.
- [51] D. Harel, *First-Order Dynamic Logic*, vol. 68 of LNCS. Heidelberg: Springer, 1979.
- [52] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic*. MIT Press, 2000.
- [53] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Cham: Springer, 2018.
- [54] S. M. Loos and A. Platzer, “Differential refinement logic,” in *LICS (M. Grohe, E. Koskinen, and N. Shankar, eds.)*, (New York, NY, USA), pp. 505–514, ACM, 2016.
- [55] E. Prebet and A. Platzer, “Uniform substitution for differential refinement logic,” in *IJCAR (C. Benzmüller, M. J. Heule, and R. A. Schmidt, eds.)*, vol. 14740 of LNCS, (Cham), pp. 196–215, Springer, 2024.
- [56] A. A. Martinez, R. Majumdar, I. Saha, and P. Tabuada, “Automatic verification of control system implementations,” in *Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, Scottsdale, Arizona, USA, October 24-29, 2010* (L. P. Carloni and S. Tripakis, eds.), (New York, NY, USA), pp. 9–18, ACM, 2010.
- [57] W. Chiang, M. Baranowski, I. Briggs, A. Solovye, G. Gopalakrishnan, and Z. Rakamaric, “Rigorous floating-point mixed-precision tuning,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (G. Castagna and A. D. Gordon, eds.), (New York, NY, USA), pp. 300–315, ACM, 2017.
- [58] N. Damouche and M. Martel, “Mixed precision tuning with salsa,” in *Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018, Porto, Portugal, July 29-30, 2018* (L. Gomes, A. Ahrens, C. Benavente-Peces, and M. S. Obaidat, eds.), pp. 185–194, SciTePress, 2018.
- [59] S. Bessaï, D. Ben Khalifa, H. Benmaghnia, and M. Martel, “Fixed-point code synthesis based on constraint generation,” in *Design and Architecture for Signal and Image Processing - 15th International Workshop, DASIP 2022, Budapest, Hungary, June 20-22, 2022, Proceedings* (K. Desnos and S. A. Pertuz, eds.), vol. 13425 of LNCS, (Cham), pp. 108–120, Springer, 2022.
- [60] A. Zeller, “Yesterday, my program worked. today, it does not. why?,” in *Software Engineering - ESEC/FSE’99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings* (O. Nierstrasz and M. Lemoine, eds.), vol. 1687 of LNCS, (Heidelberg), pp. 253–267, Springer, 1999.
- [61] 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* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of LNCS, (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.
- [62] N. S. Bjørner and L. Nachmanson, “Arithmetic solving in Z3,” in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I* (A. Gurfinkel and V. Ganesh, eds.), vol. 14681 of LNCS, (Cham), pp. 26–41, Springer, 2024.
- [63] G. Kremer, A. Reynolds, C. W. Barrett, and C. Tinelli, “Cooperating techniques for solving nonlinear real arithmetic in the cvc5 SMT solver (system description),” in *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*

- (J. Blanchette, L. Kovács, and D. Pattinson, eds.), vol. 13385 of *LNCS*, (Cham), pp. 95–105, Springer, 2022.
- [64] W. R. Inc., “Mathematica, Version 14.1,” Champaign, IL, 2024.
- [65] M. König, A. W. Bosman, H. H. Hoos, and J. N. van Rijn, “Critically assessing the state of the art in neural network verification,” *J. Mach. Learn. Res.*, vol. 25, pp. 12:1–12:53, 2024.
- [66] C. Brix, M. N. Müller, S. Bak, T. T. Johnson, and C. Liu, “First three years of the international verification of neural networks competition (VNN-COMP),” *Int. J. Softw. Tools Technol. Transf.*, vol. 25, no. 3, pp. 329–339, 2023.
- [67] C. Brix, S. Bak, T. T. Johnson, and H. Wu, “The fifth international verification of neural networks competition (VNN-COMP 2024): Summary and results,” *CoRR*, vol. abs/2412.19985, 2024.
- [68] J. Jeannin, K. Ghorbal, Y. Kouskoulas, A. C. Schmidt, R. W. Gardner, S. Mitsch, and A. Platzer, “A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system,” *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 6, pp. 717–741, 2017.
- [69] M. Wu, J. C. Rosenberg, and N. Fulton, “A formally verified plasma vertical position control algorithm,” in *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings* (M. H. ter Beek and D. Nickovic, eds.), vol. 12327 of *LNCS*, (Cham), pp. 170–188, Springer, 2020.
- [70] Y. Selvaraj, W. Ahrendt, and M. Fabian, “Formal development of safe automated driving using differential dynamic logic,” *IEEE Trans. Intell. Veh.*, vol. 8, no. 1, pp. 988–1000, 2023.
- [71] A. Platzer and E. M. Clarke, “The image computation problem in hybrid systems model checking,” in *Hybrid Systems: Computation and Control, 10th International Workshop, HSCC 2007, Pisa, Italy, April 3-5, 2007, Proceedings* (A. Bemporad, A. Bicchi, and G. C. Buttazzo, eds.), vol. 4416 of *LNCS*, (Heidelberg), pp. 473–486, Springer, 2007.
- [72] N. Fulton, S. Mitsch, J. Quesel, M. Völz, and A. Platzer, “KeYmaera X: an axiomatic tactical theorem prover for hybrid systems,” in *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings* (A. P. Felty and A. Middeldorp, eds.), vol. 9195 of *LNCS*, (Cham), pp. 527–538, Springer, 2015.
- [73] A. Kabra, S. Mitsch, and A. Platzer, “Verified train controllers for the federal railroad administration train kinematics model: Balancing competing brake and track forces,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4409–4420, 2022.
- [74] A. Platzer and J. Quesel, “European train control system: A case study in formal verification,” in *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009, Proceedings* (K. K. Breitman and A. Cavalcanti, eds.), vol. 5885 of *LNCS*, pp. 246–265, Springer, 2009.
- [75] R. Bohrer, Y. K. Tan, S. Mitsch, A. Sogokon, and A. Platzer, “A formal safety net for waypoint-following in ground robots,” *IEEE Robotics Autom. Lett.*, vol. 4, no. 3, pp. 2910–2917, 2019.
- [76] E. Prebet, S. Teuber, and A. Platzer, “Verification of autonomous neural car control with KeYmaera X,” in *Rigorous State-Based Methods - 11th International Conference, ABZ 2025, Düsseldorf, Germany, Proceedings* (M. Leuschel and F. Ishikawa, eds.), vol. 15728 of *LNCS*, Springer, 2025.
- [77] A. Kabra, J. Laurent, S. Mitsch, and A. Platzer, “CESAR: control envelope synthesis via angelic refinements,” in *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I* (B. Finkbeiner and L. Kovács, eds.), vol. 14570 of *LNCS*, pp. 144–164, Springer, 2024.
- [78] S. Teuber, S. Mitsch, and A. Platzer, “samysweb/NCubeV: v0.9,” Oct. 2024.
- [79] A. Biere, “PicoSAT essentials,” *J. Satisf. Boolean Model. Comput.*, vol. 4, no. 2-4, pp. 75–97, 2008.
- [80] J. E. Holland, M. J. Kochenderfer, and W. A. Olson, “Optimizing the next generation collision avoidance system for safe, suitable, and acceptable operational performance,” *Air Traffic Control Quarterly*, vol. 21, no. 3, pp. 275–297, 2013.
- [81] M. J. Kochenderfer, J. E. Holland, and J. P. Chrysanthacopoulos, “Next generation airborne collision avoidance system,” *Lincoln Laboratory Journal*, vol. 19, no. 1, pp. 17–33, 2012.
- [82] C. Huang, J. Fan, X. Chen, W. Li, and Q. Zhu, “POLAR: A polynomial arithmetic framework for verifying neural-network controlled systems,” in *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings* (A. Bouajjani, L. Holík, and Z. Wu, eds.), vol. 13505 of *LNCS*, (Cham), pp. 414–430, Springer, 2022.
- [83] N. Noroozi, P. Karimaghaee, F. Safaei, and H. Javadi, “Generation of Lyapunov functions by neural networks,” in *Proceedings of the World Congress on Engineering*, vol. 2008, 2008.
- [84] S. M. Richards, F. Berkenkamp, and A. Krause, “The Lyapunov neural network: Adaptive stability certification for safe learning of dynamical systems,” in *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*, vol. 87 of *Proceedings of Machine Learning Research*, pp. 466–476, PMLR, 2018.
- [85] A. Peruffo, D. Ahmed, and A. Abate, “Automated and formal synthesis of neural barrier certificates for dynamical models,” in *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 I* (J. F. Groote and K. G. Larsen, eds.), vol. 12651 of *LNCS*, (Cham), pp. 370–388, Springer, 2021.
- [86] A. Abate, D. Ahmed, M. Giacobbe, and A. Peruffo, “Formal synthesis of Lyapunov neural networks,” *IEEE Control. Syst. Lett.*, vol. 5, no. 3, pp. 773–778, 2021.
- [87] 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 *HSCC '21: 24th ACM International Conference on Hybrid Systems: Computation and Control, Nashville, Tennessee, May 19-21, 2021* (S. Bogomolov and R. M. Jungers, eds.), (New York, NY, USA), pp. 24:1–24:11, ACM, 2021.
- [88] P. Habeeb, L. Gupta, and P. Prabhakar, “Approximate conformance checking for closed-loop systems with neural network controllers,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 4322–4333, 2024.
- [89] M. K. Büning, P. Kern, and C. Sinz, “Verifying equivalence properties of neural networks with relu activation functions,” in *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings* (H. Simonis, ed.), vol. 12333 of *LNCS*, pp. 868–884, Springer, 2020.
- [90] B. Paulsen, J. Wang, and C. Wang, “Reludiff: Differential verification of deep neural networks,” *Proceedings - International Conference on Software Engineering*, pp. 714–726, 2020. arXiv: 2001.03662 ISBN: 9781450371216.
- [91] Y. Zhang, Z. Zhao, G. Chen, F. Song, M. Zhang, T. Chen, and J. Sun, “QVIP: an ilp-based formal verification approach for quantized neural networks,” in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 82:1–82:13, ACM, 2022.
- [92] P. Huang, H. Wu, Y. Yang, I. Daukantas, M. Wu, Y. Zhang, and C. W. Barrett, “Towards efficient verification of quantized neural networks,” in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada* (M. J. Wooldridge, J. G. Dy, and S. Natarajan, eds.), pp. 21152–21160, AAAI Press, 2024.
- [93] A. Adgé, D. Ben Khalifa, and M. Martel, “Fast and efficient bit-level precision tuning,” in *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings* (C. Dragoi, S. Mukherjee, and K. S. Namjoshi, eds.), vol. 12913 of *LNCS*, (Cham), pp. 1–24, Springer, 2021.
- [94] A. Kumar, V. Seshadri, and R. Sharma, “Shiftry: RNN inference in 2KB of RAM,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 182:1–182:30, 2020.
- [95] E. Park, D. Kim, and S. Yoo, “Energy-efficient neural network accelerator based on outlier-aware low-precision computation,” in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018* (M. Annavaram, T. M. Pinkston, and B. Falsafi, eds.), pp. 688–698, IEEE Computer Society, 2018.
- [96] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, “Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network,” in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los*

- Angeles, CA, USA, June 1-6, 2018 (M. Annavaram, T. M. Pinkston, and B. Falsafi, eds.), pp. 764–775, IEEE Computer Society, 2018.
- [97] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, “DRQ: dynamic region-based quantization for deep neural network acceleration,” in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*, pp. 1010–1021, IEEE, 2020.
 - [98] D. B. Khalifa and M. Martel, “Rigorous floating-point to fixed-point quantization of deep neural networks on STM32 micro-controllers,” in *2024 10th International Conference on Control, Decision and Information Technologies (CoDIT)*, pp. 1201–1206, 2024.
 - [99] F. de Dinechin, “Reflections on 10 years of flopoco,” in *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019* (N. Takagi, S. Boldo, and M. Langhammer, eds.), pp. 187–189, IEEE, 2019.

Can Large Language Models Autoformalize Kinematics?

Aditi Kabra^{id*}, Jonathan Laurent^{id†}, Sagar Bharadwaj^{id*},
Ruben Martins^{id*}, Stefan Mitsch^{id‡}, André Platzer^{id†}

^{*}Carnegie Mellon University, Pittsburgh, USA

[†]Karlsruhe Institute of Technology, Karlsruhe, Germany

[‡]DePaul University, Chicago, USA

{akabra, skalasib, rubenm}@cs.cmu.edu, jonathan.laurent@kit.edu, smitsch@depaul.edu, platzer@kit.edu



Abstract—Autonomous cyber-physical systems like robots and self-driving cars could greatly benefit from using formal methods to reason reliably about their control decisions. However, before a problem can be solved it needs to be stated. This requires writing a formal physics model of the cyber-physical system, which is a complex task that traditionally requires human expertise and becomes a bottleneck.

This paper experimentally studies whether Large Language Models (LLMs) can automate the formalization process. A 20 problem benchmark suite is designed drawing from undergraduate level physics kinematics problems. In each problem, the LLM is provided with a natural language description of the objects’ motion and must produce a model in differential game logic (dGL). The model is (1) syntax checked and iteratively refined based on parser feedback, and (2) semantically evaluated by checking whether symbolically executing the dGL formula recovers the solution to the original physics problem. A success rate of 70% (best over 5 samples) is achieved. We analyze failing cases, identifying directions for future improvement. This provides a first quantitative baseline for LLM-based autoformalization from natural language to a hybrid games logic with continuous dynamics.

Index Terms—hybrid systems, synthesis, verification, counter-example, large language models

I. INTRODUCTION

Formal methods seem especially useful in the safety-critical but mathematically complex domain of cyber-physical systems (CPSs), i.e., systems like robots and planes where discrete software interacts with the continuous dynamics of the real world. However, a bottleneck preventing broader industry adoption is the challenge of writing a formalization for these systems [22]. Unlike program verification, where a mathematically meaningful object to verify already exists in the form of a program, for CPSs, we must first create a formal model of the physical system and environment. This process is notoriously difficult, time-consuming, and error-prone.

Can large language models (LLMs) help autoformalize physical systems? They have shown autoformalization abilities for mathematics [13], [26] and CPS contracts [2], [19]. This paper presents a first, experiment-focused exploration of the ability of LLMs to autoformalize the underlying physics problems, faithfully preserving their exact continuous dynamics and discrete transitions in differential game logic (dGL) [20].

We propose 20 benchmarks derived from hard undergraduate kinematics problems [9], [14], [15]. We find that, supported by few-shot prompting and parser feedback, OpenAI o3 has a top-5 accuracy of 70% (success rate on selecting the best out of 5 attempts per benchmark). The main cause of failure is checker limitations for the most complex problems.

Our benchmarks and pipeline are available as an online artifact at <https://doi.org/10.1184/R1/28934195>. Evaluating autoformalization of a physical system is challenging because every problem has many valid formal representations, and conversely, models can be incorrect for many subtle reasons. To automatically assess the semantic proximity of generated formal models to the original physics problem, we provide a checker that:

- 1) performs *symbolic execution* over the generated formal model, examining whether the expected symbolic solution to the original physics problem is recovered.
- 2) ensures the model is not in stasis by requiring that a minimum number of variables are mutated, indicating they are subject to physical effects.

Recovering the expected solution to the original natural language problem is a strong indication that the formalization models the problem correctly, though the checker may sometimes reject answers that are in principle correct but missed some implicit assumptions that the original solution made.

Autoformalization of physical models can have tremendous impact on reliable autonomy. Unlike in mathematics, validating a hypothesized CPS formalization is possible with comparisons between real measurements and predicted measurements during test runs [16]. Easy formalization would allow every modular component to be modeled formally, with all assumptions explicit. Verification could lead to safer systems and synthesis could make control system design easier while ensuring no edge cases are missed relative to the formal model.

II. OVERVIEW

This section explains by example what the process of autoformalizing a physics problem involves. Consider the following physics question [9, problem 1.2], which will serve as a running example:

“A point traversed half the distance with a velocity v_0 . The remaining part of the distance was covered with velocity v_1 for half the time, and with velocity v_2 for the other half of the time. Find the mean velocity of the point averaged over the whole time of motion.”

Solution: “ $v_{\text{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$ ”

The physical motion described here is hybrid, with continuous dynamics (motion at velocity v_0 , v_1 , and finally v_2) as well as discrete transitions (changing velocity under specified conditions). Additionally, there are non-trivial conditions describing *when* the discrete transitions happen. We seek to symbolically model this motion formally.

A canonical way to express control problems is via hybrid games [7], [25]. Hybrid games are commonly formalized in two ways: via hybrid automata [7] and via logic [20]. In this paper, we target the logic formalization as it is closer to code, which LLMs have been more exposed to and trained for. *Differential game logic* (dGL) is a logic expressing two-player hybrid games with a relatively complete axiomatization [20]. A detailed and gradual introduction can be found in the literature [21]. Model 1 shows an example of a dGL hybrid game which formalizes the running example.

Model 1 Running Example: Finding Mean Velocity

setup	1	$\langle x := 0; d_{\text{avg}} := 0;$
phase 0	2	$\{x' = v_0, d'_{\text{avg}} = v_{\text{avg}}\}; d_h := x;$
phase 1	3	$t := 0; \{x' = v_1, d'_{\text{avg}} = v_{\text{avg}}, t' = 1\};$
phase 2	4	$t_h := t; \{x' = v_2, d'_{\text{avg}} = v_{\text{avg}}, t' = 1\};$
transitions	5	$?t = 2t_h; ?x = 2d_h$
win when	6	$\rangle d_{\text{avg}} = x$

In this game, a player (canonically called Angel) runs physical dynamics per the problem specification, moving the point’s position x with velocity v_0 in phase 0 on Line 2, with velocity v_1 in phase 1 on Line 3, and with velocity v_2 in phase 2 on Line 4. Angel is forced to transition between phases under the exact conditions specified by the problem because of *tests* (assertions) on Line 5. These assertions use *auxiliary variables* t_h to keep track of the time elapsed in phase 1, t to keep track of time elapsed in phase 1 and phase 2 combined, and d_h to keep track of half the distance covered. On Line 5, test $?x = 2d_h$ ensures that the first phase transition occurred when half the distance was covered. The test $?t = 2t_h$ ensures that the second transition happened at half the total run time of phases 1 and 2. If Angel fails either test condition, she immediately loses the game.

Notice that there is a variable v_{avg} representing average velocity, and Line 6 says that Angel will win the game only if v_{avg} was correct, as checked using auxiliary variable d_{avg} , which tracks the displacement covered by a particle traveling at v_{avg} through all phases. The value that v_{avg} must hold for Angel to win matches the solution to the original problem, $v_{\text{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$. In this example, there happens

to be no adversarial dynamics (Angel is the only player making decisions). However, in problems with adversarial nondeterminism such as unpredictable environmental factors or controller latency, a second, symmetric, adversarial player (canonically called Demon) models the situation by resolving nondeterminism with the goal to make Angel lose. Adversarial dynamics are also useful for optimization problems, e.g., “choose the launch angle that minimizes drift”, which can be modeled as Angel and Demon competing to find this angle.

Having formalized the problem as a dGL hybrid game, we next describe how symbolic execution enables a check for whether the model aligns with the natural language question. For the fragment of dGL where most of the chosen benchmark problems lie (loop-free, polynomial solutions), symbolic execution is decidable and existing work [11] describes how to perform it. Backwards symbolic execution over a game lets us compute the conditions under which Angel (or dually, Demon) can win the game. For example, symbolic execution of the subgame $\langle \text{Line 5} \rangle d_{\text{avg}} = x$ at the end of Model 1 evaluates to $x = d_{\text{avg}} \wedge t = 2t_h \wedge x = 2d_h$ which is precisely the weakest precondition starting from which Angel passes the tests of Line 5 and ends the game in a state where she wins. Consider the precondition ϕ of Line 1 computed in this way. Angel should be able to win the game exactly when v_{avg} is set to the correct solution, i.e., $v_{\text{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2} \leftrightarrow \phi$. Such a check is implemented using Z3 and Mathematica.

Observe that even a minor modeling mistake can result in this check failing. In fact, symbolically executing Model 1 reveals that it is actually equivalent to $\top$. It suffers from a subtle exploit where Angel can always run each phase for 0 time, in which case $x_{\text{avg}} = x = 0$ regardless of what v_{avg} is set to. The problem is that the model lets Angel choose the distance that the particle will travel, as her choice at Line 2 determines the value of half-distance d_h , which in turn determines the distance that phases 1 and 2 must cover. The fix is simple: the assignment $d_h := x$ on Line 2 should be turned to a test $?d_h = x$. The automated check was able to catch this difference and is a strong signal of correctness, especially in combination with manual review. The automated checker has a second test to ensure the system is not in stasis, ruling out empty games that exploit the symbolic execution test by setting the win condition directly to the desired solution. For this example, the second test checks via static analysis that at least two variables are written to, since *any* reasonable model will at least modify the variables representing the particle’s position and time. The checker is provided with the minimum number of variables that must be written to and the expected solution for every benchmark problem.

Finally, we see the benefit of formal modeling when the model with Line 2 corrected is still not equivalent to the expected solution. The problem is that the textbook solution is correct *only under some unstated assumptions*: distance d is non-zero, velocity v_0 is positive, and $v_1 + v_2$ is also positive. Finally, Model 2, makes these assumptions. Under the assumptions of Model 2, Line 1, the modal formula from

Model2, Line2 to Line7 is equivalent to $\frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$. In unverified or informal physics, subtle modeling glitches are widespread. As a human, it is easy to neglect a measure zero case like $d_h = 0$ during an intermediate calculation, but formalization forces this case to be explicit.

Model 2 Repaired Model: Finding Mean Velocity

assum2	1	$(d_h > 0 \wedge v_0 > 0 \wedge v_1 + v_2 > 0) \rightarrow$
setup	2	$\langle x := 0; d_{\text{avg}} := 0;$
phase 0	3	$\{x' = v_0, d'_{\text{avg}} = v_{\text{avg}}\}; ?d_h = x;$
phase 1	4	$t := 0; \{x' = v_1, d'_{\text{avg}} = v_{\text{avg}}, t' = 1\};$
phase 2	5	$t_h := t; \{x' = v_2, d'_{\text{avg}} = v_{\text{avg}}, t' = 1\};$
transitions	6	$?t = 2t_h; ?x = 2d_h$
win when	7	$\rangle d_{\text{avg}} = x$

This example begins to demonstrate why formalizing a hybrid physical problem is *not* a simple translation task. Introducing the right auxiliary variables, absent in the specification, is often necessary for a clean, tractable formalization. All the pieces of the model must fit together precisely, e.g., even making a weak inequality strict can transform a correct model into one admitting no runs [24]. Correctness requires careful attention to the subtleties of the problem specification, e.g., benchmark problem 10 (available in the online artifact [10]) has one variable representing a *displacement* and another, representing a *distance*. The former is a signed number while the latter must be modeled as non-negative. Models must prune mathematically degenerate cases using background knowledge of the physical world, e.g., by eliminating imaginary roots of polynomials computing real quantities. Additionally, there are many ways to model the same problem, and the right choice of coordinate system or frame of motion can be crucial to writing a model for which symbolic reasoning is tractable.

Given the complexity of the task, it is unclear whether LLMs can succeed at all, making a careful evaluation necessary. We create a benchmark suite drawing from textbooks and problem sets [9], [14], [15], choosing tricky problems with varying structures to systematically identify the strengths and weaknesses of LLMs on autoformalization.

III. METHODOLOGY

The autoformalization system, shown by Fig. 1, takes as input the natural language specification of the physical problem to model, and produces either a correct dGL formalization or a failure (no formalization found that passed the checker). It uses two types of LLM queries: one is *Propose Formalization*, and the second one is *Revise Formalization*. The artifact [10] shows all the prompt templates used.

The autoformalization system first queries the LLM to propose the formalization using the *Propose Formalization* query, providing up to four solved examples as part of the prompt. It provides an LLM with the natural language description of the problem and asks it to produce a dGL formula, including in the prompt common dGL syntax mistakes and their remedies.

An example of a problem description the prompt would use based on the running example is

“An object A starts from rest and travels a distance d. For the first half of the distance, it travels with velocity v0. For the second half of the distance, it spends half the time traveling with velocity v1 and half the time traveling with velocity v2. Another object B tracks A. Its motion has the same duration, starting, and ending points as A, but it travels with uniform velocity vavg. Let vavg remain a free variable.”

The prompt also includes upto four solved examples. The returned formal model is passed to the KeYmaera X dGL parser for a syntax check.

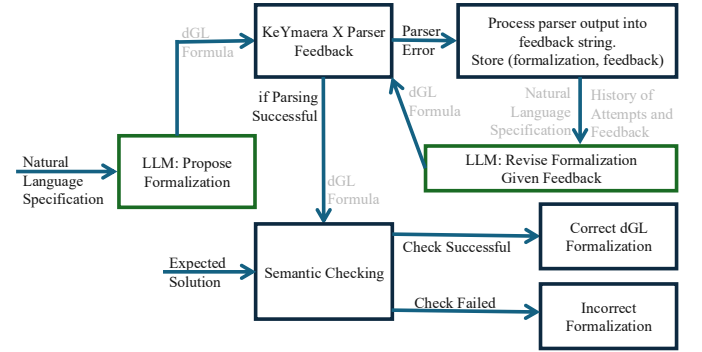


Fig. 1. Autoformalization pipeline

If the syntax check fails, then the parser output is processed into short feedback strings. An example of a parser feedback string is

“The input formula contains an unsupported Unicode character (possibly {bad_char}). Use only ASCII characters.”

Where *bad_char* is a parameter extracted from parser output. The incorrect formalization along with the feedback is then appended to a list of previous failed formalization attempts.

The LLM is prompted to repair the formalization given past attempts and feedback using the *Revise Formalization* query. The prompt first provides up to two examples of repair. Then it provides the natural language question, and finally past incorrect proposals along with the feedback indicating why these proposals were wrong. The LLM replies with a new, repaired dGL game, which is again passed to the parser for checking. This syntactic repair loop is allowed to run at most three times, after which autoformalization fails. A full implementation, along with examples, is available in the online artifact [10].

If the syntax check succeeds, then the dGL formula is passed onto the semantic evaluation system. This system takes as input the expected symbolic solution to the problem along with the proposed dGL formula. For our running example, the expected solution consists of the formula $v_{\text{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$, to

test against for equivalence, and minimum expected number of variables that are written to, 2. If the checker finds that the dGL formula is equivalent to the expected solution under the initial assumptions of the formula and has enough variable writes, then it declares that the formalization is acceptable. The pipeline is implemented using Delphyne [12] and KeYmaera X [6], which in turn uses Z3 [4] and Mathematica.

IV. RELATED WORK

Given a natural language description, formal physics models are generally written manually. Prior work has studied the use of structured natural language to perform formalization by rule-based transformations [1], [18], but this limits flexibility in input specification. Recent progress in NLP (LLMs) raises the prospect of automating formalization directly from unstructured natural language.

The use of LLMs for autoformalizing mathematics has received significant research attention [13], [26], including for purely continuous systems with partial differential equation [5], [8]. Recent work also uses LLMs to synthesize contracts for cyber-physical systems [2], [17], [19]. However, we focus on LLM based autoformalization for *physics models* of hybrid systems, with discrete transitions and continuous dynamics faithfully represented, which to the best of our knowledge has not yet been studied. Compared to general mathematics and LTL specifications, we expect this domain to be more data-scarce, with few examples of differential dynamic logic system formalizations and even fewer examples of differential game logic games available on the Internet. Additionally, we expect CPS model formalization to have a more operational flavor, modeling physical realities required to make a system work rather than abstract concepts that can be simplified for theoretical convenience (mathematics) or focused on only output behavior (LTL contracts). This difference makes it worth investigating the hybrid systems modeling domain.

V. EVALUATION

We propose 20 benchmarks derived from challenging problems in undergraduate physics textbooks and problem sets [9], [14], [15]. The online artifact [10] lists them. Phrasing is changed to focus on modeling the physical situation described in the problem (rather than on solving for a variable) and the numbers are changed to induce rational answers to avoid failures of the automated check over numerical imprecision. Each benchmark is accompanied by an expected symbolic solution that the model should be equivalent to and the minimum expected number of variables that must be written to. Four additional problems, separate from the benchmarks, serve as examples for few-shot prompting.

We run the LLM autoformalization system using OpenAI GPT 4o, GPT 4.1 (best available OpenAI non-reasoning model) and OpenAI o3 (best available OpenAI reasoning model) on the benchmark set, getting successively improving results. Outcomes are classified into 4 buckets:

- 1) *Success*: Correct, checker-certified formalization.

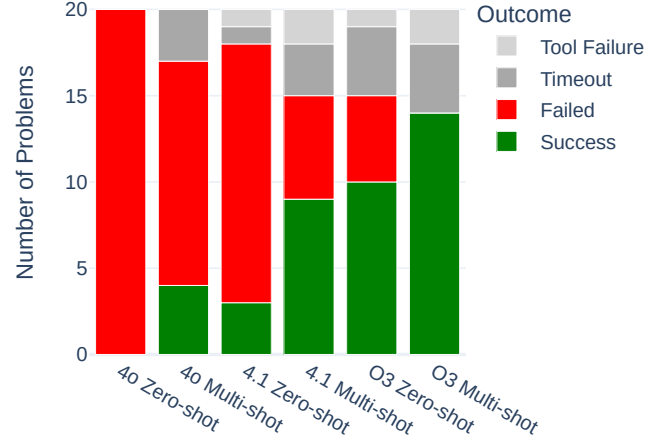


Fig. 2. Outcomes of autoformalization by GPT 4o, GPT-4.1 and o3, with zero-shot and multi-shot prompting.

- 2) *Failed*: autoformalization failed either the syntax or semantic check.
- 3) *Timeout*: The solver timed out after 3 minutes of symbolic execution. Symbolic execution uses real quantifier elimination, which is doubly exponential [3], and can result in timeouts.
- 4) *Tool Failure*: The solver encountered an expression that it could not automatically symbolically execute (e.g., ODE with a square roots on the right hand side).

Fig. 2 shows the outcome. The autoformalization system is sampled five times independently with temperature 1, and the best outcome is reported (where tool failure and solver timeout are considered better than failure). For each sample, the LLM is given three chances to repair syntax if the KeYmaera X parser reported an error. We evaluate two different prompts, one using few-shot prompting considering four examples (*multi-shot*) and another one without examples (*zero-shot*).

GPT 4o without few-shot prompting fails on every benchmark. Few-shot prompting significantly improves performance for all models. With few-shot prompting o3 has a success rate of 70%, which is higher than GPT 4.1’s 45% and GPT 4o’s 20%. Our results support that the improvements and the introduction of reasoning LLM models significantly increase their success rate on mathematical tasks, enabling strong performance in the autoformalization of hybrid games.

VI. DISCUSSION

Fig. 3 shows the outcome of each problem by model. We can observe that models found the same problems hard. There are six problems that no model solves. For these, the best performing model produces a formalization that could potentially be correct but is too complex for the checker to verify. The problems can be classified into three groups that explain where their complexity arises from.

- 1) *Optimization Problems*: Problem 5 and 20.

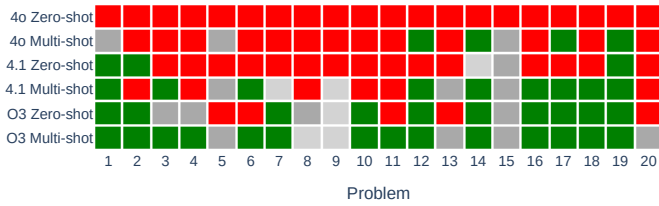


Fig. 3. Problem outcome by model.

- 2) Non-polynomial continuous dynamics: Problem 8 and 9.
- 3) Complicated dynamics: Problem 13 and 15.

The first category, for example, includes problem 5, which is about identifying the launch angle of a boat that minimizes drift, modeled by adversarial Angel trying to pick a better angle than Demon, who must set the optimal angle to win. The interplay of the two players leads to combinatorial branching over possible sequence of events, while continuous dynamics occur under constraints preventing control blocking. These constraints make symbolic reasoning about the differential equations particularly computationally expensive.

An example of the second category is problem 8, which is about drones arranged in an equilateral triangle that cyclically pursue each other with constant radial velocity, collapsing inwards in increasingly rapid revolutions, till they crash. Their dynamics, often modeled with a non-polynomial square root expression on the right side of an ODE, cannot be handled automatically by the tool. Problem 13 is an example from the third category. It has three phases of dynamics, including one where acceleration has a continuous dependence on time. Such a differential equation is more expensive to symbolically evaluate, making a timeout the likely outcome.

Sometimes a problem fundamentally requires capabilities that a solver lacks, and the solver must be extended. Other times a difficult but crucial part of formalizing problems is choosing the right abstractions and representations that let proofs succeed. For the six failing problems, the solution likely lies in a combination of these methods. The solver must be extended to better automatically handle, for instance, non-polynomial dynamics by using continuous invariant generation [23]. Complementing this, the LLM should be guided to use tricks like polar coordinates and auxiliary variables to rewrite dynamics in friendlier forms, and to also provide checking hints such as continuous invariants.

Unsolved benchmarks provide a roadmap for improvement in both these directions. Additionally, the rapid improvement in success rate displayed by better LLM models suggests that autoformalization of hybrid games will continue to improve with advancements in LLMs.

VII. CONCLUSION AND FUTURE WORK

This work experimentally studies the ability of LLMs to formally model physical behavior given a natural language specification. The results (70% success rate) are encouraging,

and suggest that CPS model autoformalization using LLMs could be an impactful direction for future research.

Beyond the broader adoption of formal methods in offline controller design, physics autoformalization enables a new class of applications, where autonomous systems autoformalize unfamiliar situations they face in the open world and derive formally justified control decisions to respond to them. For such applications, where the *physical environment* must be formalized at runtime, the input to autoformalization would be vision and sensor data rather than a natural language description. LLM autoformalization from such multimodal input data provides an interesting future research direction.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation (NSF) under Award Number CCF2427581, DARPA under Agreement FA8750-24-9-1000, and by an Alexander von Humboldt Professorship.

REFERENCES

- [1] Gustavo Carvalho, Ana Cavalcanti, and Augusto Sampaio. Modelling timed reactive systems from natural-language requirements. *Formal Aspects of Computing*, 28(4):725–765, 2016.
- [2] Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu 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*, pages 15880–15903, Singapore, December 2023. Association for Computational Linguistics.
- [3] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, 5(1/2):29–35, 1988.
- [4] 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, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [5] Mengge Du, Yuntian Chen, Zhongzheng Wang, Longfeng Nie, and Dongxiao Zhang. Large language models for automatic equation discovery of nonlinear dynamics. *Physics of Fluids*, 36(9), 2024.
- [6] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völpl, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In *CADE*, pages 527–538, 2015.
- [7] Thomas A. Henzinger, Benjamin Horowitz, and Rupak Majumdar. Rectangular hybrid games. In *Proceedings of the 10th International Conference on Concurrency Theory, CONCUR '99*, page 320–335, Berlin, Heidelberg, 1999. Springer-Verlag.
- [8] Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2678–2710, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.
- [9] I. E. Irodov. *Problems in General Physics*. Mir Publishers, 1988.
- [10] Aditi Kabra, Jonathan Laurent, Sagar Bharadwaj, Ruben Martins, Stefan Mitsch, and Andre Platzer. Artifact for Can Large Language Models Autoformalize Kinematics? 2025. https://kilthub.cmu.edu/articles/software/Artifact_for_Can_Large_Language_Models_Autoformalize_Kinematics_/28934195. DOI: 10.1184/R1/28934195.
- [11] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and André Platzer. CESAR: Control envelope synthesis via angelic refinements. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 14570 of *LNCS*, pages 144–164. Springer, 2024.
- [12] Jonathan Laurent and André Platzer. Oracular programming: A modular foundation for building llm-enabled software, 2025.

- [13] Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Fan Yang, Xian Zhang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 53598–53625. Curran Associates, Inc., 2024.
- [14] Samuel J. Ling, Jeff Sanny, and William Moebs. *University Physics: Volume 1*. OpenStax, Houston, TX, September 19 2016.
- [15] MIT OpenCourseWare. 8.01SC Classical Mechanics Problem Set 1. https://ocw.mit.edu/courses/8-01sc-classical-mechanics-fall-2016/resources/mit8_01f16_pset1_new/, 2016. Fall 2016, undergraduate level.
- [16] Stefan Mitsch and André Platzer. ModelPlex: verified runtime validation of verified cyber-physical system models. *Formal Methods Syst. Des.*, 49(1-2):33–74, 2016.
- [17] Daniel Neider and Rajarshi Roy. *What Is Formal Verification Without Specifications? A Survey on Mining LTL Specifications*, pages 109–125. Springer Nature Switzerland, Cham, 2025.
- [18] Bruno Oliveira, Gustavo Carvalho, Mohammad Reza Mousavi, and Augusto Sampaio. Simulation of hybrid systems from natural-language requirements. In *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pages 1320–1325, 2017.
- [19] Jiayi Pan, Glen Chou, and Dmitry Berenson. Data-efficient learning of natural language to linear temporal logic translators for robot task specification. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11554–11561, 2023.
- [20] André Platzer. Differential game logic. *ACM Trans. Comput. Log.*, 17(1):1:1–1:51, 2015.
- [21] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, 2018.
- [22] Kristin Yvonne Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 8–26, Cham, 2016. Springer International Publishing.
- [23] Andrew Sogokon, Stefan Mitsch, Yong Kiam Tan, Katherine Cordwell, and André Platzer. Pegasus: Sound continuous invariant generation. *Form. Methods Syst. Des.*, 58(1):5–41, 2022. Special issue for selected papers from FM’19.
- [24] Yong Kiam Tan and André Platzer. Switched systems as hybrid programs**this research was sponsored by the afosr under grant number fa9550-16-1-0288. the first author was also supported by a *star, singapore. *IFAC-PapersOnLine*, 54(5):247–252, 2021. 7th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2021.
- [25] Claire J. Tomlin, John Lygeros, and Shankar Sastry. A game theoretic approach to controller design for hybrid systems. *Proc. IEEE*, 88(7):949–970, 2000.
- [26] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS ’22*, Red Hook, NY, USA, 2022. Curran Associates Inc.

Towards SMT Solver Stability via Input Normalization

Daneshvar Amrollahi^{*}, Mathias Preiner^{*}, Aina Niemetz^{*}, Andrew Reynolds[†],
Moses Charikar^{*}, Cesare Tinelli[†], Clark Barrett^{*}

^{*}Stanford University, Stanford, USA

{daneshvar, preiner, niemetz, mooses, barrett}@cs.stanford.edu

[†]The University of Iowa, Iowa City, USA

{andrew-reynolds, cesare-tinelli}@uiowa.edu



Abstract—In many applications, SMT solvers are utilized to solve similar or identical tasks over time. Significant variations in performance due to small changes in the input are not uncommon and lead to frustration for users. This sort of *stability* problem represents an important usability challenge for SMT solvers. We introduce an approach for mitigating the stability problem based on normalizing solver inputs. We show that a perfect normalizing algorithm exists but is computationally expensive. We then describe an approximate algorithm and evaluate it on a set of benchmarks from related work, as well as a large set of benchmarks sampled from SMT-LIB. Our evaluation shows that our approximate normalizer reduces runtime variability with minimal overhead and is able to normalize a large class of mutated benchmarks to a unique normal form.

I. INTRODUCTION

SMT solvers are used to solve a large variety of problems in academia and industry [4], [14], [21]. As these solvers are integrated into more and more workflows, they are increasingly used in situations where there are many calls to a solver with identical or similar queries. For example, a software verification tool may run a regression suite nightly to check that some software meets its specification, and for the most part, this nightly run does not differ much, if at all, from the previous night's run. A common pain point in such situations is that SMT solver performance can vary significantly, even if there are only minor changes. This has been termed the *stability* problem: queries that are semantically similar or identical may require vastly different amounts of time to solve. Or, even worse, some minor changes may result in a formerly solved query not being solved at all.

As part of an NSF-supported project [3], we spoke with a large number of SMT stakeholders and cataloged and ranked over 100 different issues, based on how often they came up in interviews. In this ranking, instability was ranked as the second-highest area of concern (behind only requests for better diagnostic output when the solver is unable to solve a problem). The instability concern was also highlighted in a keynote talk by Neha Rungta at CAV 2022 [21] on the use of SMT solvers at Amazon Web Services and has been

identified by another senior manager at Amazon as one of two top priorities for improving their SMT solving workflows [13].

There are two kinds of changes that can introduce instability: changes to the input and changes to the solver. While both are important, this paper focuses on the first: *changes to the input*. In other words, we are interested in reducing the sensitivity of solving time to minor changes in an input formula, especially if those changes are semantics-preserving.

Two common misconceptions about stability are worth noting. First, instability should not be confused with poor solver performance. Improving solver performance is an important and worthy goal; however, it is orthogonal to addressing the stability problem. Our focus is on making solver performance more *consistent* in the face of mutations to its input. Importantly, many users in our study reported that they would welcome *any* improvement to stability, even if it came at the cost of some degradation in performance. The other misconception derives from not recognizing the connection between stability and computational complexity. Because the SMT problem is NP-hard (or worse, depending on the theory being used), solvers try to guide worst-case exponential algorithms in such a way that solutions are found quickly when possible. However, even a small change in the input can cause the exploration of a different search path, which can result in an exponentially worse (or better) runtime. This is similar to the well-known *butterfly effect* in chaos theory. Despite this problem, as we show in this paper, there are steps that can be taken to *reduce* instability.

SMT solvers typically already use deterministic data structures and algorithms to eliminate easily avoidable sources of instability. In a sequential execution setting, the remaining source of variability is their search heuristics, primarily their branching heuristics, which typically break ties based on the *order* in which declarations and assertions appear in the input problem. Our key idea is to reduce this variability by using the *structure of the input problem* to determine the order of declarations and assertions. Our approach uses the principle of *normalization*: we attempt to map classes of semantically equivalent inputs to the same normal form. Note that if perfect normalization could be achieved in this setting, it would result in perfect stability with respect to the semantically equivalent

This work was supported in part by the National Science Foundation (award #2303489), a gift from Amazon Web Services, and the Stanford Center for Automated Reasoning.

input classes, as the input to the core solver would be the same in every case. We present an approach aimed at getting closer to that ideal without introducing too much overhead. A notable feature of our approach is that it is agnostic to the underlying solver. It can thus be used to improve the stability of any SMT solver with respect to changes in the input.

We consider normalization with respect to a set of basic, semantics-preserving transformations: (i) reordering of assertions; (ii) reordering of operands of commutative operators; (iii) reordering and renaming of user-defined symbol declarations; and (iv) replacing anti-symmetric operators by their converse. These transformations are representative of the kinds of changes that happen in the real-world scenarios motivating our work. For example, in a program verification workflow, if a function is moved or renamed, this could result in a reordering of declarations and assertions, a different name for a user-defined symbol, or both. These transformations are also a superset of the solver input transformations considered in previous work on *measuring* instability [26]. We will refer to these transformations as *mutations* and to inputs that have been transformed this way as *mutated* inputs. We address the following research questions:

- 1) Is it possible to design a normalizing algorithm that utilizes these mutations to map all mutated variants to a single unique normal form?
- 2) If such an algorithm exists, what is its time complexity?
- 3) How closely can an efficient algorithm approximate the ideal algorithm?

After covering some background in Section II, we formalize the problem in Section III and answer the first two questions, showing that such an algorithm does indeed exist but is as hard as graph isomorphism. We provide an answer to the third question in the remainder of the paper. Section IV introduces an algorithm that approximates the ideal algorithm, and Section V presents an evaluation of our implementation, showing that it significantly improves SMT solver stability on benchmarks from the Mariposa project [26] and from the SMT-LIB benchmark library [20]. Finally, Section VI concludes.

Related work: The importance of the issue of stability in SMT solving has been raised in other work [8], [10], [12], [17]. Dodds [8], highlights the problem of proof fragility under changes in verification tools. Ferraiuolo et al. [10] mention proof instability as *the most frustrating recurring problem*, especially when proof complexity increases as a result of reasoning about procedures with many instructions and complex specifications. In Hawblitzel et al. [12], verification instability is observed in large formulas and non-linear arithmetic due to different options for applicable heuristics. Leino & Pit-Claudel [17] identify *matching loops*—caused by certain forms of quantifiers that lead an SMT solver to repeatedly instantiate a limited set of quantified formulas—as a key factor contributing to instability in verification times, and describe techniques to detect and prevent them.

More relevant to our work is the work of Zhou et al. [26], [25]. In [26], they pioneer an effort to detect and quantify instability and introduce a tool for this task called Mariposa.

They show that mainstream SMT solvers such as Z3 [7] and cvc5 [1] exhibit instability on a set of F^* [22] and Dafny [16] benchmarks [5], [10], [11], [18], [19], [23]. They consider benchmark-modifying mutations, specifically symbol renaming and assertion reordering, as well as solver-modifying mutations, via the use of different random seeds, and employ a statistical approach to identify instability arising from these mutations. We include an evaluation of our technique on their benchmarks but focus on benchmark-modifying mutations only as we aim to improve stability with respect to input changes.

In [25], Zhou et al. identify irrelevant context in a query as one source of instability and propose a novel approach to filter out such context to improve solver stability. This approach is complementary to our own, and combining the two is an interesting direction for future work.

Also closely related is the work of Weber [24], which introduces a normalizer designed to reverse the effects of mutations applied to *scramble* benchmarks for the SMT-COMP competition at the time. The normalizer introduced in [24] was specifically designed to expose a weakness in the scrambling algorithm where symbols were renamed but not reordered. This weakness makes it possible to achieve perfect normalization efficiently, as demonstrated by Weber’s algorithm. However, in the presence of the more expressive and realistic mutations we consider here, the normalizer from [24] performs poorly, as we show in Section V.

II. BACKGROUND

We work in the context of many-sorted logic (e.g., [9]), where we assume an infinite set of variables of each sort and the usual notions of signatures, terms, formulas, assignments, and interpretations. We assume a signature Σ consisting of sort symbols and sorted function symbols. It is convenient to assume that Σ has a distinguished sort `Bool`, for the Booleans, and to represent relation symbols as function symbols whose return sort is `Bool`. We also assume the signature includes equality. Symbols in Σ are partitioned into *theory symbols* (e.g., $=, \wedge, \vee, +, -, 0, 1$) and *user-defined symbols* (e.g., f, g, x, y). We assume some *background theory* that restricts the theory symbols to have fixed interpretations, whereas the interpretation of user-defined symbols is left unrestricted.

We represent formulas as finite *sequences* of symbols in prefix notation, where each symbol is either a theory symbol or a user-defined symbol. This causes no ambiguity when each symbol has a fixed arity. If S is a sequence $\langle s_1, \dots, s_n \rangle$, we write $|S|$ to denote n , the length of the sequence, and S_i to denote the i^{th} element of the sequence. We write $s \in S$ to mean that s occurs in the sequence S , and write $S \circ S'$ for the sequence obtained by appending S' at the end of S . We write $user(S)$ to mean the sequence obtained by deleting all theory symbols in S , e.g., $user(\langle x, +, y, -, x \rangle) = \langle x, y, x \rangle$. We denote the set of integers between m and n inclusive, where $n \geq m$, as $[m, n]$. And we abbreviate $[1, n]$ as $[n]$.

An input problem in the SMT-LIB 2.6 format [2] is shown in Figure 1. We use this as a running example throughout

```

(set-logic QF_UFLIA)

(declare-fun f (Int) Int) (declare-const v Int)
(declare-const w Int) (declare-const x Int)
(declare-const y Int) (declare-const z Int)

(assert (>= (+ (f x) y) (- v 12)))
(assert (< (+ x y) (* x z)))
(assert (>= (+ (f y) x) (- w 12)))
(assert (< (+ y x) (* x x)))
(assert (< (+ x y) (* y y)))
(assert (< (+ y x) (* y v)))

(check-sat)

```

Fig. 1: Running Example.

```

(assert (>= (+ (f x) y) (- v 12)))
(assert (< (+ x y) (* x z)))

(assert (> (* u4 u2) (+ u2 u3)))
(assert (>= (+ (g u2) u3) (- u1 12)))

```

Fig. 2: Two different mutated versions of the same assertions.

the paper. The example includes arithmetic theory symbols, a user-defined function f , and user-defined constants v, w, x, y , and z . We will refer to the sequence of six assertions in the running example as $\langle \beta_1, \dots, \beta_6 \rangle$. Figure 2 shows two different representations of the first two assertions in the example: the first is from the example and the second is a mutation of it. In particular, the order of the assertions has been swapped, the operands of the $*$ operator have been reordered, the user-defined symbols have been renamed (with w, x, y , and z renamed to $u1$ through $u4$, respectively, and f renamed to g), and the assertion using $<$ has been rewritten to use $>$.

III. FORMALIZATION

Consider again the first two assertions from the running example. Here, the user-defined symbols are $\{f, x, y, v, z\}$, and the theory symbols are $\{>=, <, +, -, *, 12\}$. When written as a sequence in prefix notation, assertion β_1 is $\langle >=, +, f, x, y, -, v, 12 \rangle$. Similarly, β_2 is $\langle <, +, x, y, *, x, z \rangle$.

Recall that mutations include four operations: (i) reordering assertions; (ii) reordering operands of commutative operators; (iii) reordering and renaming symbols; and (iv) replacing anti-symmetric operators. We defer (iv) to the end of this section, as it can be easily handled separately. To formalize the others, we introduce some definitions.

Definition III.1 (Shuffle Set). The *shuffle set* of a sequence A is defined as $S(A) = \{A' \mid A' \text{ is a permutation of } A\}$.

For example, $S(\langle \beta_1, \beta_2 \rangle) = \{\langle \beta_1, \beta_2 \rangle, \langle \beta_2, \beta_1 \rangle\}$.

Definition III.2 (Commutative Reordering Set). Let α be a formula, possibly containing (binary) commutative operators. The *commutative reordering set* of α is defined as $C(\alpha) = \{\alpha' \mid \alpha' \text{ is the result of swapping the operands of zero or more of these commutative operators in } \alpha\}$. For a sequence A of n formulas, $C(A)$ is the set of all sequences $\langle \alpha'_1, \dots, \alpha'_n \rangle$ where for each $i \in [n]$, $\alpha'_i \in C(\alpha_i)$. If X is a set (containing either formulas or sequences of formulas), then $C(X) = \{x' \mid x' \in C(x) \text{ for some } x \in X\}$.

For example, $C(\beta_1) = \{\beta_1, \langle >=, +, y, f, x, -, v, 12 \rangle\}$. Note that there is only one entry in $C(\beta_1)$ besides β_1 itself, because $+$ is the only commutative operator in β_1 . On the other hand, there are four elements in $C(\beta_2)$, since β_2 has two commutative operators, $+$ and $*$. Consequently, $C(\langle \beta_1, \beta_2 \rangle)$ has eight elements, representing all combinations of one formula each from $C(\beta_1)$ and $C(\beta_2)$.

Definition III.3 (Pattern). Let α be a formula. The *pattern* of α , written $P(\alpha)$, is a sequence of the same length as α , defined for each $i \in [|\alpha|]$ by

$$P(\alpha)_i = \begin{cases} \alpha_i & \text{if } \alpha_i \text{ is a theory symbol,} \\ @1 & \text{if } \alpha_i \text{ is the first user-defined symbol in } \alpha \\ P(\alpha)_j & \text{if } \alpha_i \text{ is a user-defined symbol and} \\ & \text{there exists } j \in [i-1] \text{ with } \alpha_j = \alpha_i, \\ @k & \text{otherwise, where } k = 1 + \\ & |\{\alpha_j \mid j \in [i-1], \alpha_j \text{ user-defined}\}|. \end{cases}$$

For convenience, we assume that each symbol $@k$ is a fresh constant¹ of the same sort as the symbol it is replacing, so that if α is well sorted, then so is $P(\alpha)$. We call *pattern symbols* the introduced symbols starting with “@”, and *patterns* the sequences in the co-domain of P . We define a total order $\prec$ on patterns to be the lexicographic order induced by some total order on formula symbols.² For our example assertions, we have $P(\beta_1) = \langle >=, +, @1, @2, @3, -, @4, 12 \rangle$ and $P(\beta_2) = \langle <, +, @1, @2, *, @1, @3 \rangle$.

We lift this notation to sequences and sets of sequences. To explain how, we need two more definitions.

Definition III.4. Given a sequence of formulas $A = \langle \alpha_1, \dots, \alpha_n \rangle$, the *conjoining* of A , $Conj(A)$ is the formula $\langle \wedge \rangle \circ \alpha_1 \circ \alpha_2 \circ \dots \circ \alpha_n$. Similarly, given a formula $\langle \wedge \rangle \circ \alpha_1 \circ \alpha_2 \circ \dots \circ \alpha_n$, where α_i is a formula for $i \in [n]$, the *unconjoining* of α , written $Unconj(\alpha)$ is the formula sequence $\langle \alpha_1, \dots, \alpha_n \rangle$.

For a sequence A of formulas, we define $P(A) = Unconj(P(Conj(A)))$. If X is a set containing either formulas or sequences of formulas, then $P(X) = \{P(x) \mid x \in X\}$. We similarly lift $\prec$ to sequences of formulas: if A and A' are sequences of patterns, then $A \prec A'$ iff $\alpha_1 \circ \dots \circ \alpha_{|A|} \prec \alpha'_1 \circ \dots \circ \alpha'_{|A'|}$. Next, we define renaming and normalization.

Definition III.5 (Renaming). Let V be a set of variables. A *renaming* R is an injective function from pattern symbols to V . For a formula α , $R(\alpha)$ is defined to be a sequence of the same size as α , defined as follows:

$$R(\alpha)_i = \begin{cases} \alpha_i & \text{if } \alpha_i \text{ is a theory symbol,} \\ R(\alpha_i) & \text{if } \alpha_i \text{ is a pattern symbol.} \end{cases}$$

¹The SMT-LIB 2.6 standard reserves symbols starting with @ for internal use by solvers, so this assumption is a reasonable one.

²An obvious choice for this order (and the one we use) is the lexicographic order on the string representations of theory and pattern symbols.

For a sequence of formulas $A = \langle \alpha_1, \dots, \alpha_n \rangle$, $R(A) = \langle R(\alpha_1), \dots, R(\alpha_n) \rangle$. If X is a set of formulas or sequences of formulas, then $R(X) = \{R(x) \mid x \in X\}$.³

Definition III.6 (Normalizing Function). A function N from sequences of formulas to sequences of formulas is said to be *normalizing* if, for every sequence A of formulas:

- 1) $N(A) = R(A')$ for some $A' \in P(C(S(A)))$ and some renaming R ; and
- 2) if $M_1 = R_1(M'_1)$ and $M_2 = R_2(M'_2)$, with R_1, R_2 renamings and with $M'_1, M'_2 \in P(C(S(A)))$, then $N(M_1) = N(M_2)$.

We can now introduce our first research question: do normalizing functions exist? We show that the question can be answered affirmatively.

Definition III.7 (Normalizing Function). Let $\mathcal{N}$ be defined as follows. For every sequence of formulas A , $\mathcal{N}(A)$ is the $\prec$ -minimal element of $P(C(S(A)))$.

Lemma III.8. *Let A be a sequence of formulas. If $A_R = R(A_P)$, where $A_P \in P(C(S(A)))$ and R is a renaming, then $P(C(S(A_R))) = P(C(S(A)))$.*

Proof. First, note that for any formula sequence A , we have $R(P(C(S(A)))) = C(S(R(P(A))))$. This is because the first transformation generates a set of equivalent formula sequences and then renames the symbols, while the second renames the symbols and then generates a set of equivalent formula sequences. However, these two operations are independent, so their combined result is the same, regardless of their order.

Now, let $A_R = R(P(A'))$ for some $A' \in C(S(A))$. We have that $P(C(S(A_R))) = P(C(S(R(P(A'))))) = P(R(P(C(S(A'))))) = P(C(S(A')))$, since computing the pattern of a renaming of a pattern is just the same as computing the pattern. It remains to show that $P(C(S(A')) = P(C(S(A)))$. Since $A' \in C(S(A))$, A can be obtained from A' by swapping zero or more commutative operators and permuting the order of the formulas in A' . But then, any element of $C(S(A))$ can be obtained from A' by swapping commutative operators and permuting the order, so $C(S(A'))$ is just the same as $C(S(A))$. Thus, $P(C(S(A'))) = P(C(S(A)))$. $\square$

Lemma III.9. *For any formula A , $P(C(S(A)))$ has a unique minimal element with respect to $\prec$.*

Proof. When comparing two elements of $P(C(S(A)))$, each of which is a sequence of patterns, we concatenate the patterns together and compare them with $\prec$. Since $\prec$ is a total order, one of them is always smaller. Thus $P(C(S(A)))$ has a minimum element. $\square$

Theorem III.10. *Function $\mathcal{N}$ is normalizing.*

Proof. Notice that $\mathcal{N}(A) = \mathbb{R}(A')$, where $\mathbb{R}$ is the identity function and A' is the $\prec$ -minimal element of $P(C(S(A)))$.

³When representing an assertion sequence as an SMT-LIB 2.6 script, we declare user-defined symbols in lexicographic order, so a renaming can affect the order of declarations.

It is not hard to see that $\mathbb{R}$ is also a renaming, so the first requirement is met. Now, let $M_1 = R_1(M'_1)$ and $M_2 = R_2(M'_2)$, with $M'_1, M'_2 \in P(C(S(A)))$, where R_1, R_2 are renamings. By Lemma III.8, $P(C(S(M_1))) = P(C(S(A))) = P(C(S(M_2)))$. But $P(C(S(A)))$ has a unique minimal element, A' by Lemma III.9. Thus, $\mathcal{N}(M_1) = A' = \mathcal{N}(M_2)$. $\square$

A. Complexity

We have shown the existence of a normalizing function. The next question is whether normalization can be done efficiently. An informal argument that computing the normalization of an arbitrary set of formulas is at least as hard as graph isomorphism can be found in Lavrov [15], and a formal proof can be found in Weber [24]. The question of whether graph isomorphism can be solved in polynomial time is a long-standing open problem.

Theorem III.11. *Let N be a normalizing function. Then, computing $N(A)$ for an arbitrary A is as hard as solving graph isomorphism.*

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two undirected graphs. Assume without loss of generality that $V_1 \cap V_2 = \emptyset$. For each $v \in \{V_1 \cup V_2\}$, let $u(v)$ map v to some unique user-defined symbol (i.e., u is injective). Now, define $A_i = \{\langle f, x \rangle \mid x \in V_i\} \cup \{\langle =, x, y \rangle \mid \exists (v, v') \in E_i. \{x, y\} = \{u(v), u(v')\}\}$, where f is some Boolean predicate. Note that G_i can be recovered from A_i , simply by creating a vertex for every user-defined symbol appearing as an argument of f in A_i and then adding an edge between $u^{-1}(x)$ and $u^{-1}(y)$ whenever the formula $\langle =, x, y \rangle$ appears in A_i . Furthermore, shuffling the formulas in A_i , permuting the order of the operands in equalities (the only commutative operator appearing in A_i), or renaming the user-defined symbols does not change the structure of the graph being represented. Thus, all elements of $P(C(S(A_i)))$ represent isomorphic graphs.

Now, suppose $N(A_1) = N(A_2)$. For $i \in [1, 2]$, we know that $N(A_i) = R(A'_i)$ for some $A'_i \in P(C(S(A_i)))$. Furthermore, because renamings are injective, $A'_1 = P(R(A_1)) = P(N(A_1)) = P(N(A_2)) = P(R(A_2)) = A'_2$. But for $i \in [1, 2]$, the graph represented by A_i is isomorphic to the graph represented by A'_i . Thus the graph represented by A_1 , namely G_1 , is isomorphic to the graph represented by A_2 , which is G_2 .

On the other hand, suppose G_1 and G_2 are isomorphic. Let $h : V_1 \rightarrow V_2$ be the isomorphism function for the graph vertices. Let R be a renaming such that $\forall v. R(P(v)) = h(v)$. Applying this renaming to A_1 must be equivalent (modulo order) to A_2 . In other words, $A_1 \in R(P(S(A_2)))$. Then, because N is normalizing, we must have $N(A_1) = N(A_2)$. Thus, computing N is at least as difficult as graph isomorphism. $\square$

Note that the use of transformation C is not essential in the proof above. This means that normalizing just the result of shuffling and renaming is already as hard as graph isomorphism.

```
(assert (>= (+ (f x) y) (- v 12)))
(assert (< (+ x y) (* x z)))
```

```
(assert (>= (+ (f x) y) (- v 12)))
(assert (> (* x z) (+ x y)))
```

Fig. 3: Original (top) and normalized (bottom) assertions.

B. Anti-symmetric Operators

As mentioned above, mutations can randomly replace anti-symmetric operators with their dual operator. For example, $(\langle + x y \rangle (* x z))$ could be changed to $(\langle > (* x z) (+ x y) \rangle)$. In general, we allow mutations that transform expressions of the form $A \text{ op } B$ into $B \text{ op}' A$, where (op, op') pairs include: $(\langle, \rangle)$, $(\geq, <)$, $(\geq, \leq)$, $(\text{bvugt}, \text{bvult})$, $(\text{bvuge}, \text{bvule})$, and so on.

A normalization algorithm can easily handle anti-symmetric operators, simply by choosing one representative operator for each pair and forcing all assertions to use only the chosen operators. For example, if we choose the first operator in each pair listed above, then Figure 3 shows the result of normalizing the first two assertions in our running example. Notice that the first assertion is unchanged because it is already using the chosen operator.

C. Relation to Permutation Groups, Graph Isomorphism, and Symmetry Breaking

Our formalization draws on well-established concepts such as permutation groups and graph isomorphism. The relevant permutation groups arise as automorphism groups of the structures we consider, that is, the set of all renamings preserving structure. Graph isomorphism provides the framework for determining when two structures are equivalent under such permutations and captures the underlying symmetry inherent in the problem. However, unlike standard symmetry breaking, which prunes a search space by imposing syntactic orderings, our goal is to rewrite all isomorphic structures, possibly using different names or expressions, into a common normal form. This requires normalization guided by the underlying structure.

IV. APPROXIMATING A NORMALIZATION ALGORITHM

As a first step towards a general practical algorithm for normalization, we describe a heuristic procedure designed to handle two of the four mutations: shuffling and renaming. We leave the handling of the other operations to future work. We expect support for normalizing antisymmetric operator replacement to be straightforward (as described above), while normalizing commutative operand swapping will be more challenging. Note that shuffling and renaming are also the two operations used to mutate benchmarks in the Mariposa work (the closest related work) [26]. We describe our algorithm at a conceptual level here, and discuss several optimizations necessary to make it work well in practice in Section IV-A. Our algorithm consists of three steps: (i) Sorting the assertions; (ii) Renaming all symbols; and (iii) Sorting the assertions

```
(assert (< (+ x y) (* x z)))
(assert (< (+ y x) (* y v)))

(assert (< (+ y x) (* x x)))
(assert (< (+ x y) (* y y)))

(assert (>= (+ (f x) y) (- v 12)))
(assert (>= (+ (f y) x) (- w 12)))
```

Fig. 4: Assertions sorted by pattern.

again. In the rest of this section, we discuss these steps in detail.

Sorting the assertions. Step one is to sort the assertions. The challenge is to do this in a way that does not depend on the names of user-defined symbols. The key idea is to use patterns. In particular, to order assertions α and α' , we can compare $P(\alpha)$ and $P(\alpha')$ using the $\prec$ order. For instance, considering again assertions from our running example, β_5 is $\langle \langle, +, x, y, *, y, y \rangle$, and β_6 is $\langle \langle, +, y, x, *, y, v \rangle$, so we have $P(\beta_5) = \langle \langle, +, @1, @2, *, @2, @2 \rangle$ and $P(\beta_6) = \langle \langle, +, @1, @2, *, @1, @3 \rangle$. The first difference in the patterns is in the sixth position. Assuming $@1$ comes before $@2$ in the ordering, we have that $P(\beta_6) \prec P(\beta_5)$, so we can conclude that β_6 should be placed before β_5 .

Note, however, that it is possible for two formulas to have the same pattern. Thus, after sorting according to patterns, we obtain an ordered list of equivalence classes $EC_1, \dots, EC_n$ with the following features:

- 1) α and α' belong to the same equivalence class iff $P(\alpha) = P(\alpha')$.
- 2) $\alpha \in EC_i$ and $\alpha' \in EC_j$ where $i < j$ iff $P(\alpha) < P(\alpha')$.

The next question is whether we can easily order the assertions belonging to the same equivalence class. We give an efficient approximation method. For this, we need the notions of role and super-pattern.

Definition IV.1 (Role). The *role* of a symbol s in a formula α , denoted $\text{role}(s, \alpha)$, is 0 if $s \notin \alpha$ and is the index of the earliest occurrence of s in $\text{user}(\alpha)$, otherwise. The role of s in a set of formulas is the multiset consisting of all the roles played by s in the formulas in the set.

For example, consider the role of y in β_5 . First of all, we compute $\text{user}(\beta_5)$, which is $\langle x, y, y, y \rangle$. We can then see that y occurs first at the second position, so $\text{role}(y, \beta_5) = 2$. Similarly, $\text{role}(x, \{\beta_4, \beta_5, \beta_6\}) = \{1, 1, 2\}$.

Definition IV.2 (Super-pattern). The *super-pattern* of a symbol s over a sequence X of sets $X_1, \dots, X_n$, denoted $SP(s, X)$, is the sequence of roles of the symbol in each set: $SP(s, X) = \langle \text{role}(s, X_1), \text{role}(s, X_2), \dots, \text{role}(s, X_n) \rangle$.

Let EC be a sequence of formula equivalence classes. The super-pattern of EC captures the role of a symbol across all equivalence classes, while treating the formulas in each equivalence class as unordered.

To illustrate, recall the example from Figure 1. Figure 4 shows the result of sorting the assertions by pattern, resulting in three equivalence classes, each separated by an

empty line. The patterns of the equivalence classes in $EC = \{EC_1, \dots, EC_3\}$, from top to bottom, are as follows:

$$\begin{aligned} EC_1: & \langle <, +, @1, @2, *, @1, @3 \rangle \\ EC_2: & \langle <, +, @1, @2, *, @2, @2 \rangle \\ EC_3: & \langle >=, +, @1, @2, @3, -, @4, 12 \rangle \end{aligned}$$

Now, suppose we want to order the formulas in EC_3 . We compare the super-patterns of the first different pair of user-defined symbols, in this case, x and y . The roles of x throughout the equivalence classes are:

$$\begin{aligned} \text{role}(x, EC_1) &= \{\text{role}(x, \beta_2), \text{role}(x, \beta_6)\} = \{1, 2\} \\ \text{role}(x, EC_2) &= \{\text{role}(x, \beta_4), \text{role}(x, \beta_5)\} = \{1, 2\} \\ \text{role}(x, EC_3) &= \{\text{role}(x, \beta_1), \text{role}(x, \beta_3)\} = \{2, 3\} \end{aligned}$$

Therefore, applying the definition of super-pattern for x yields $SP(x, EC) = \{\{1, 2\}, \{1, 2\}, \{2, 3\}\}$. It is not hard to see that the super-pattern for y is the same, so the two assertions cannot be distinguished by looking at x and y . The next pair of different user-defined variables also consists of x and y . However, the last pair is v and w . Following the same process, we find that $SP(v, EC) = \{\{0, 4\}, \{0, 0\}, \{0, 4\}\}$ and $SP(w, EC) = \{\{0, 0\}, \{0, 0\}, \{0, 4\}\}$. Now, all we need is a way to order different super-patterns.

Definition IV.3 (Integer multiset order). Given multi-sets of integers m_1 and m_2 , $m_1 < m_2$ iff the sequence of nondecreasing elements of m_1 is lexicographically smaller than the sequence of nondecreasing elements of m_2 .

Definition IV.4 (Super-pattern order). For super-patterns s_1 and s_2 , $s_1 < s_2$ iff s_1 comes before s_2 when compared using the lexicographic order induced by the integer multiset order.

Thus, when comparing super-patterns, we compare the entries in the sequences one by one using the integer multiset order. The first two entries in $SP(v, EC)$ and $SP(w, EC)$ are $\{0, 4\}$ for v and $\{0, 0\}$ for w . Because $\langle 0, 0 \rangle < \langle 0, 4 \rangle$, we can conclude that $SP(w, EC) < SP(v, EC)$. Thus, we should reverse the order of assertions in the last equivalence class. Similarly, by computing super-patterns for z and v , we can see that we should keep the order of the assertions in the first equivalence class.

It is possible for all of the super-patterns of corresponding symbols in two assertions in the same equivalence class to be the same. In this case, our heuristic fails to distinguish the assertions, and the assertion order is left unchanged. For example, for EC_2 , the only user-defined symbols available for comparison are x and y , and they have the same super-pattern. Thus, we leave these assertions in their original order for now. Algorithm 1 shows the full algorithm for ordering two assertions.

Renaming all symbols. After sorting the assertions according to Algorithm 1, we rename all the symbols in the assertions. We use a renaming $\mathbb{R}$ that maps a variable whose pattern symbol is $@k$ to X_k . More precisely, if A is the sequence of assertions after sorting, we replace A with $\mathbb{R}(P(A))$. Figure 5 shows the assertions from our running example after sorting and renaming.

Algorithm 1 Algorithm for ordering assertions A and B given EC , a sequence of equivalence classes (with respect to pattern equality) of assertions.

```

if  $P(A) \neq P(B)$  then
  return  $P(A) \prec P(B)$ 
end if
for  $i \leftarrow 1$  to  $|user(A)|$  do
   $u \leftarrow user(A)_i$ 
   $v \leftarrow user(B)_i$ 
  if  $u = v$  then
    continue
  end if
  if  $SP(u, EC) \neq SP(v, EC)$  then
    return  $SP(u, EC) < SP(v, EC)$ 
  end if
end for
return inconclusive

```

```

(assert (< (+ X1 X2) (* X1 X3)))
(assert (< (+ X2 X1) (* X2 X4)))

(assert (< (+ X2 X1) (* X1 X1)))
(assert (< (+ X1 X2) (* X2 X2)))

(assert (>= (+ (X5 X1) X2) (- X4 12)))
(assert (>= (+ (X5 X2) X1) (- X6 12)))

```

Fig. 5: Assertions after sorting and renaming.

Sorting the assertions again. After renaming, there is one more step that can improve the normalizer. It is based on the observation that within an equivalence class, different assertion orders are possible, depending on the initial order, when all symbols in a pair of assertions have the same super-patterns. This can be partially addressed by lexicographically sorting each equivalence class after renaming. This ensures that if we have two benchmarks for which the first two steps produce the same set of assertions, but in different orders, then these two benchmarks will be normalized the same way.

Looking again at Figure 5, we see that assertions in the first and last equivalence classes are already in sorted order. However, the assertions in equivalence class 2 should be reordered. Recall that in the previous step, we did not have a way to order these assertions, but now there is an unambiguous order for them. It is important to note that our algorithm does not guarantee the normalization property. The reason for this incompleteness is that when assertions cannot be distinguished by super-patterns, there can be different normal forms for benchmarks, even if one is a mutation of the other by shuffling and renaming. Nevertheless, our algorithm works well in practice, as we show in the next section.

A. Code Optimizations

Some of the benchmarks in our benchmark sets are extremely large, with sizes up to hundreds of megabytes and with up to hundreds of thousands of user-defined symbols and assertions. In this section, we discuss three optimizations

that are crucial for the scalability of our algorithm on such benchmarks.

a) Pattern Compression: As presented in Section III, formulas are sequences whose size corresponds to the size of their abstract syntax tree. However, internally, SMT solvers represent formulas as directed acyclic graphs (DAGs), because they often share subterms. In other words, the internal representation effectively achieves common subexpression elimination. The result can be exponentially more concise. It is thus essential to also incorporate this kind of sharing in our representation of patterns. In our implementation, we represent a pattern actually as an index into a dictionary of the subterms in the tree representation of the pattern. For example, the formula $\langle +, *, x, x, *, x, x \rangle$ would be represented as the index **2** into the dictionary $(\mathbf{1} : \langle *, x, x \rangle, \mathbf{2} : \langle +, \mathbf{1}, \mathbf{1} \rangle)$.

b) Super-pattern Computation: A naive way to compute the super-pattern for a symbol s is to traverse all of the assertions and look up the role of s in every assertion. However, this approach is not scalable for large benchmarks. To address this issue, we perform a one-time indexing pass over the assertions, to create arrays for each symbol s containing pointers to only the assertions in which s appears. This way, we only need to traverse those assertions when calculating the super-pattern for s . This optimization reduces the number of lookups from hundreds of millions to hundreds of thousands on some problematic benchmarks, improving the normalization time by more than an order of magnitude.

c) Super-pattern Compression: This optimization leverages the sparsity of super-patterns: in large benchmarks, each symbol typically appears in only a small subset of the assertions. Thus, the role of a symbol is likely to be 0 in most assertions. To exploit this, we represent super-patterns as sequences consisting of the non-zero role values interleaved with counts of the number of zeros.

V. EXPERIMENTS

We implemented our normalization algorithm in C++. We evaluate its effectiveness on three dimensions: *normalization effectiveness*, *stability*, and *runtime*.

We use two sets of benchmarks. The first set, *smtlib*, is obtained by randomly selecting 50 benchmarks from each family (or all of them, if there are fewer than 50) in the SMT-LIB benchmark library [20]. A family, in this context, consists of all benchmarks in a single leaf directory of the filesystem hierarchy. Note that even though our normalization algorithm as described in Section IV is, in theory, applicable to all of SMT-LIB without limitation, our current implementation does not support the normalization of algebraic datatypes due to implementation-level complexities. We thus excluded benchmarks with algebraic datatypes from our evaluation. Even with that exclusion, the *smtlib* set contains 41,166 benchmarks, from 1,581 benchmark families. We group benchmarks from this set into the (so-called) divisions used by the annual SMT-COMP solver competition [6] to keep the number of categories manageable.

The second set, *mariposa*, consists of the benchmarks used in [26] (as provided in [27]), originating from program verification projects written in Dafny [16], Serval [19], and F* [22]. The *mariposa* set contains 16,622 benchmarks and is divided into six families: Dice_F* [23] (1536), Komodo_D and Komodo_S [10] (2,054 and 773), VeriBetrKV_D [11] (5,170), VeriBetrKV_L [18] (5,334) and vWasm_F [5] (1,755).

For each benchmark in one of the two sets, we produce 10 mutations by randomly applying assertion shuffling and renaming. Our renaming uses a fixed enumeration of names (i.e., a benchmark with n user-defined symbols always uses the first n of these names), but shuffles the order in which they appear in assertions (and thus the order in which they are declared, since we declare symbols in the order in which they appear). We ran all experiments on a cluster of 48 machines with AMD Ryzen 9 7950X CPUs using a time limit of 60 seconds and a memory limit of 8GB. We used this same time and memory limit to separately limit the mutation step, the normalization step, and the solving step.⁴ We do not report results for benchmarks that timed out during mutation or normalization. For the *smtlib* set, we exclude 6 benchmarks that timed out during the mutation step and 85 that timed out during the normalization phase (of these, 58 timed out during parsing, before ever getting to the normalization algorithm), for any of the 10 mutated versions. In *mariposa*, we only exclude 2 benchmarks, both of which timed out during normalization. Overall, this resulted in 41,075 eligible benchmarks in *smtlib* and 16,620 in *mariposa*.

A. Normalization Effectiveness

In our first experiment, we evaluate how closely our implementation of an efficient normalization algorithm approximates the ideal algorithm on both benchmark sets. For this, we measure, for each benchmark: (i) the number of distinct outputs produced by our normalizer for the 10 mutations (the best possible is 1; the worst possible is 10); and (ii) the *similarity* of the 10 normalized outputs, computed as the average percentage of identical lines among all pairs of outputs. Table I shows our results. Each row starts with a category name (the name of the division or family) and a pair of numbers (x/y) denoting the number of excluded benchmarks x vs. the total number of benchmarks y in that category. We then list the number of considered benchmarks (*#Bench*, equal to $y - x$) and the average number of distinct benchmarks produced by the 10 mutations, before normalization (*pre*) and after normalization, without (*wo-SP*) and with (*w-SP*) the use of super-patterns. We next show the average and maximum runtime for the normalization algorithm. Finally, we show the average similarity among the mutated benchmarks, again comparing *pre*, *wo-SP*, and *w-SP*. Figure 6 shows histograms of the number of distinct benchmarks among the 10 mutations before and after (with super-patterns) normalization.

⁴This is long enough to be able to catch cases when the normalizer is slow and short enough to help keep a large evaluation computationally tractable. Using the same timeout for all stages also keeps things simple.

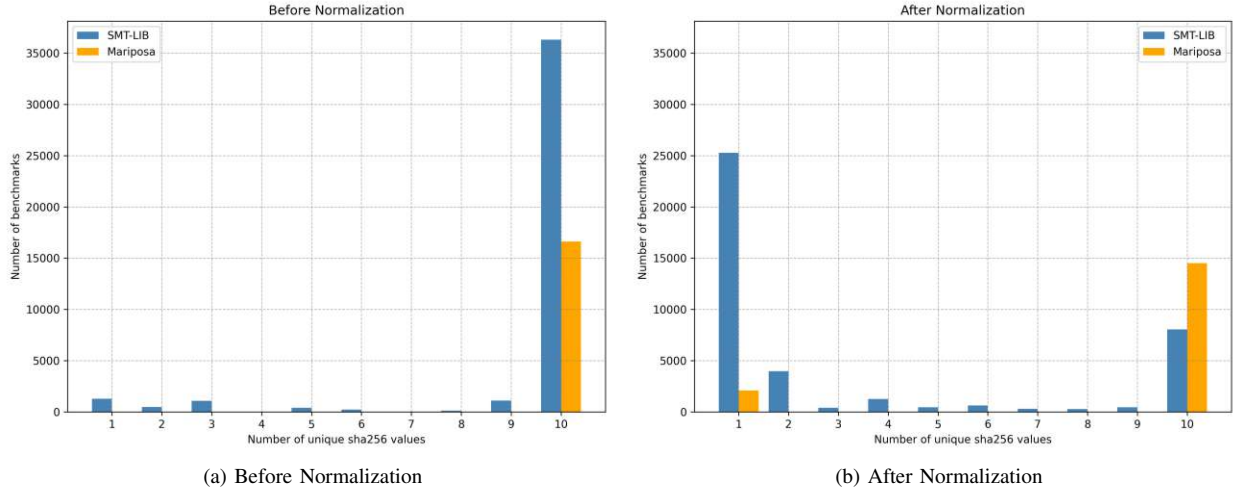


Fig. 6: Number of distinct benchmarks before and after normalization.

Division (<i>smtlib</i>)	(Excluded/Total)	#Bench	#Unique			Time (s)		Similarity (%)		
			<i>pre</i>	<i>wo-SP</i>	<i>w-SP</i>	<i>Avg</i>	<i>Max</i>	<i>pre</i>	<i>wo-SP</i>	<i>w-SP</i>
Arith	(0/883)	883	9.33	1.05	1.02	0.0010	0.1720	7.07	99.39	99.62
BitVec	(5/793)	788	9.36	1.07	1.03	0.0139	3.5230	5.81	99.01	99.48
Equality	(0/1,570)	1,570	10.00	8.48	2.79	0.0075	0.1020	0.05	47.59	98.48
Equality+LinearArith	(0/2,595)	2,595	9.99	7.34	5.35	0.0030	0.0860	1.29	65.56	86.78
Equality+MachineArith	(2/905)	903	10.00	1.25	1.22	0.0295	3.6140	0.76	97.97	98.18
Equality+NonLinearArith	(0/1,452)	1,452	9.70	2.98	1.64	0.0080	0.4980	3.14	88.89	98.48
FPArith	(0/319)	319	9.48	1.03	1.03	0.0000	0.0010	5.25	99.61	99.61
QF_Bitvec	(42/3,320)	3,278	9.55	4.94	4.78	0.7248	38.7570	4.63	74.17	79.88
QF_Equality	(0/667)	667	10.00	2.63	2.16	0.0306	1.0710	0.73	85.67	93.24
QF_Equality+Bitvec	(20/1,208)	1,188	9.87	4.49	4.44	0.1222	13.3070	2.94	71.07	74.06
QF_Equality+LinearArith	(12/812)	800	9.95	3.66	3.38	0.0590	6.6460	1.58	83.17	89.38
QF_Equality+NonLinearArith	(0/589)	589	10.00	6.73	6.22	0.2908	29.6690	1.76	71.40	85.08
QF_FPArith	(0/4,813)	4,813	9.08	2.80	2.52	0.0002	0.0290	13.17	90.93	92.30
QF_LinearIntArith	(6/10,801)	10,795	9.98	6.35	4.21	0.2728	37.2840	8.62	78.79	88.67
QF_LinearRealArith	(3/1,466)	1,463	9.95	3.72	3.26	0.2230	30.3610	6.77	91.19	93.86
QF_NonLinearIntArith	(0/653)	653	9.88	3.04	2.64	0.0752	4.5400	4.77	94.00	95.69
QF_NonLinearRealArith	(1/3,343)	3,342	9.83	1.83	1.67	0.0620	10.5110	8.42	96.05	97.76
QF_Strings	(0/4,977)	4,977	6.21	2.69	2.45	0.0004	0.0190	44.50	91.75	94.93
Family (<i>mariposa</i>)										
Dice _F	(0/1536)	1536	10.00	10.00	10.00	5.4847	12.4670	0.93	99.45	98.75
Komodo _D	(0/2054)	2054	10.00	10.00	10.00	1.2750	3.7500	0.18	34.87	88.28
Komodo _S	(2/773)	771	10.00	1.00	1.00	0.0004	0.0790	1.09	100.00	100.00
VeriBetrKV _D	(0/5170)	5170	10.00	10.00	10.00	0.0790	9.1740	0.19	18.15	87.30
VeriBetrKV _L	(0/5334)	5334	10.00	10.00	10.00	1.2533	9.6420	0.17	12.73	81.85
vWasm _F	(0/1755)	1755	10.00	3.16	3.15	0.0657	4.2050	0.01	99.53	99.52

TABLE I: Number of unique outputs, normalization time, and similarity of normalized outputs.

First, we observe that before normalization, we nearly always have 10 distinct versions of a benchmark. Exceptions (e.g., in the QF_Strings category) occur when there are not a sufficient number of user-defined symbols and assertions to create 10 distinct versions. For the *smtlib* benchmark set, normalization always significantly reduces the number of distinct versions of a benchmark. Moreover, results with super-patterns always improve over those without them—sometimes significantly (e.g., in the Equality division).

For other benchmarks, however, especially those in the QF_Equality+NonLinearArith division and in several families of the *mariposa* benchmark set, our normalizer struggles to produce a small number of distinct outputs. Investigating these

benchmarks reveals large numbers of assertions with lots of symmetry. In these cases, the super-pattern comparison fails to distinguish between assertions with identical patterns, resulting in different normal forms because of assertions appearing in a non-deterministic order. It is worth highlighting that for these benchmarks, our normalization algorithm still achieves a high level of similarity. As we discuss below, there is also an improvement in stability for these benchmarks. This suggests that full normalization is not necessary to improve stability. Improved similarity is already helpful.

As mentioned above, over all of the benchmarks, only 87 are excluded due to timeouts during normalization, and these are typically very large benchmarks where parsing alone takes

most or all of the time. For the vast majority of the remaining benchmarks, the normalization overhead is extremely low (a fraction of a second on average as shown in Table I). The aggregate overhead, computed as the sum of the normalizing time for all benchmarks divided by sum of the normalization plus solving time for all benchmarks, is less than 0.8%.

We also conducted exploratory experiments on a random subset of *smtlib* (4,461 benchmarks) to evaluate the performance of the normalizer in Weber [24] in terms of uniqueness of the normalized output. We observed that it was able to produce unique outputs for only 12% of the benchmarks, vastly underperforming in comparison to our approach which succeeded 87% of the time. This result, however, is not unexpected since the algorithm in [24] was designed for the purpose of exploiting a specific weakness in the SMT-COMP’s scrambling algorithm in use at the time rather than with general-purpose normalization in mind.

B. Stability

In our second experiment, we evaluate how our normalization algorithm affects solver stability. We use two SMT solvers: *cvc5* [1] and *Z3* [7]. These are natural choices, as both are used extensively and support a wide range of theories. We limit our evaluation to them since they are the only solvers that support all of our benchmarks. For the *mariposa* benchmark set, we limit our evaluation to *Z3*. This is because these benchmarks come from a specific use case targeting *Z3* (as observed already by Zhou et al. [26]), and most of them are unsolved by *cvc5*.

We use penalized runtime (PR-2) and Median Absolute Deviation (MAD) as metrics. PR-2 is the sum of the time taken for solved benchmarks plus a penalty equal to two times the timeout for each unsolved benchmark (timeouts, memory outs, or other errors). PR-2 thus combines elements of both total time and number of solved benchmarks into a single metric. MAD measures how much variation there is in a set of results, with lower numbers indicating less variation and higher numbers indicating more variation. Applying MAD to PR-2 scores is thus a good proxy for stability.

Results are shown in Table II for each division (in *smtlib*) and family (in *mariposa*). We report results on the benchmarks before normalization (*no norm.*) and after normalization with super-patterns (*norm.*). The column labeled *avg* contains the average PR-2 score, computed as follows. Recall that we create ten mutations for each benchmark where each mutation is based on a specific random seed. We compute the total PR-2 score for all the benchmarks for each seed separately. We then take the average of these ten PR-2 scores. The MAD score is also computed over these ten cumulative PR-2 scores, one for each seed.

We see that, for both solvers, the performance (avg. column) on normalized benchmarks is generally comparable with that of non-normalized benchmarks, showing that, on average, normalization does not appear to greatly affect runtime.

More importantly, the MAD score improves significantly in *all* cases, sometimes by more than an order of magnitude

Division (<i>smtlib</i>)	cvc5 PR-2				Z3 PR-2			
	no norm. avg. MAD	no norm. avg. MAD	norm. avg. MAD	norm. avg. MAD	no norm. avg. MAD	no norm. avg. MAD	norm. avg. MAD	norm. avg. MAD
Arith	17,161	93.5	17,264	2.9	21,532	120.6	11,346	0.3
BitVec	29,141	41.0	29,247	14.6	34,245	121.4	28,248	14.6
Equality	101,912	286.0	101,778	4.0	107,718	108.8	107,885	10.7
Equality+LinearArith	77,933	208.4	77,620	32.8	72,603	254.5	73,041	22.2
Equality+MachineArith	91,935	23.4	91,518	6.8	74,037	452.8	73,539	37.2
Equality+NonLinearArith	97,462	351.0	95,961	10.8	89,615	375.5	90,551	17.1
FPArith	18,089	49.6	17,920	49.5	14,371	143.2	14,214	28.5
QF_Bitvec	149,591	282.8	153,082	208.9	104,282	751.8	94,428	174.6
QF_Equality	2,408	49.0	2,435	3.7	1,285	64.4	1,195	1.0
QF_Equality+Bitvec	40,712	157.1	40,246	153.7	36,282	265.5	35,870	84.7
QF_Equality+LinearArith	14,927	46.4	16,070	44.9	8,278	296.4	4,862	179.5
QF_Equality+NonLinearArith	34,344	422.9	34,234	276.0	25,811	374.6	25,731	175.3
QF_FPArith	28,971	289.6	28,063	108.1	56,349	324.4	55,262	106.3
QF_LinearIntArith	242,429	402.8	244,127	148.7	151,935	606.8	132,783	345.7
QF_LinearRealArith	30,567	313.6	29,539	101.0	23,547	310.8	21,260	160.4
QF_NonLinearIntArith	34,201	615.7	34,267	26.3	23,914	194.7	22,265	6.8
QF_NonLinearRealArith	39,951	86.8	39,377	27.9	26,186	188.3	28,016	131.7
QF_Strings	34,092	341.4	33,255	121.6	56,970	321.4	56,420	158.4
Family (<i>mariposa</i>)								
Dice _F	–	–	–	–	7,531	559.3	8,951	545.3
Komodo _D	–	–	–	–	12,561	264.2	13,327	59.9
Komodo _S	–	–	–	–	3,284	79.9	3,181	17.2
VeriBetrKV _D	–	–	–	–	12,623	414.4	12,612	169.0
VeriBetrKV _L	–	–	–	–	24,889	203.6	29,971	172.7
vWasm _F	–	–	–	–	2,919	9.5	3,018	0.2

TABLE II: PR-2 and MAD scores on mutated benchmarks before and after normalization.

(e.g., in the Equality division). This strongly suggests that our normalization algorithm improves stability. Even for benchmarks for which the normalizer completely fails to produce normal forms (e.g., the *mariposa* benchmarks that always still have 10 distinct benchmarks after normalization), the stability improves, sometimes significantly. This provides promising evidence that full normalization is not required for improving stability. An improvement in similarity may be sufficient.

VI. CONCLUSION

Our normalization algorithm is a promising step towards a more stable and predictable SMT solving experience. It generally scales well and is applicable to a wide range of benchmarks and logics. We have shown that it can often produce a single unique normal form for a set of mutated benchmarks, and even when it cannot, it greatly increases the similarity of the benchmarks. We also saw that, on average, it improves stability without significantly affecting performance.

In future work, we intend to expand our normalizer to handle other mutations such as antisymmetric operator replacement and operand swapping for commutative operators. We also plan to explore whether additional normalization techniques can be used to further improve our results. Finally, we plan to investigate the use of our normalization algorithm as a preprocessing step to improve the hit rate of applications that cache formulas in order to reuse solving results.

REFERENCES

- [1] 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 - 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.
- [2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
 - [3] Clark Barrett and Cesare Tinelli. POSE: Phase II: An open-source ecosystem for the cvc5 SMT solver, 2023.
 - [4] Nikolaj S. Bjørner. SMT solvers: Foundations and applications. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 24–32. IOS Press, 2016.
 - [5] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multi-lingual software sandboxing using webassembly. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 1975–1992. USENIX Association, 2022.
 - [6] Martin Bromberger, François Bobot, and Martin Jonáš. SMT Competition (SMT-COMP) 2024, 2024.
 - [7] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
 - [8] Mike Dodds. Formally verifying industry cryptography. *IEEE Secur. Priv.*, 20(3):65–70, 2022.
 - [9] Herbert Enderton and Herbert B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.
 - [10] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305. ACM, 2017.
 - [11] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 99–115. USENIX Association, 2020.
 - [12] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 165–181. USENIX Association, 2014.
 - [13] Robert Jones. Private communication, 2024.
 - [14] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.
 - [15] Misha Lavrov. Algorithm for finding a “normal” form for a set of strings under character-mapping isomorphism? Mathematics Stack Exchange. <https://math.stackexchange.com/q/2925668> (version: 2018-09-21).
 - [16] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
 - [17] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016.
 - [18] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–28, 2022.
 - [19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019.
 - [20] Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. SMT-LIB release 2024 (non-incremental benchmarks), April 2024.
 - [21] Neha Rungta. A billion SMT queries a day (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2022.
 - [22] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.
 - [23] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. Dice*: A formally verified implementation of DICE measured boot. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1091–1107. USENIX Association, 2021.
 - [24] Tjark Weber. Scrambling and descrambling SMT-LIB benchmarks. In Tim King and Ruzica Piskac, editors, *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories, Coimbra, Portugal, July 1-2, 2016*, volume 1617 of *CEUR Workshop Proceedings*, pages 31–40, July 2016.
 - [25] Y. Zhou, J. Bosamiya, J. G. Li, M. J. H. Heule, and B. Parno. Context pruning for more robust smt-based program verification. In N. Narodytska and P. Rümmer, editors, *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024*, pages 59–69. TU Wien Academic Press, 2024.
 - [26] Yi Zhou, Jay Bosamiya, Yoshiaki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 178–188. IEEE, 2023.
 - [27] Yi Zhou, Bryan Parno, Marijn Heule, and Jay Bosamiya. Mariposa: Measuring smt instability in automated program verification [benchmarks], January 2025.

Per-Instance Subproblem Generation for Strategy Selection in SMT

Amalee Wilson^{*}, Nina Narodytska[†], Clark Barrett^{*}, and Haoze Wu[‡]

^{*}Stanford University, Stanford, USA {amalee, barrett}@cs.stanford.edu

[†]Amherst College, Amherst, USA hwu@amherst.edu

[‡]VMware Research by Broadcom, Palo Alto, USA nina.narodytska@broadcom.com

Abstract—In this paper, we investigate customizing the solving strategy for an individual SMT problem, based solely on the problem itself, without relying on any offline strategy tuning. Our key insight is to generate a set of subproblems derived from the original formula, analyze the behavior of candidate solving strategies on these smaller, representative subproblems, and predict which strategy will perform best on the original formula. We demonstrate that performance on the subproblems is frequently indicative of performance on the original formula. Additionally, we introduce a novel subproblem generation procedure that outperforms existing SMT formula partitioning techniques for the proposed workflow. Finally, we show that on a selection of SMT-LIB benchmarks, when our approach can make a prediction, it can reduce the total compute time substantially.

I. INTRODUCTION

Algorithmic tuning is a standard approach for optimizing an SMT solver’s performance on a specific problem domain. Traditionally, tuning is regarded as an *offline* process, whereby a collection of related benchmarks is used to evaluate candidate solving strategies (e.g., different solvers, different solver options). The goal is either to find the best overall strategy or to train a strategy selector that predicts the best solving strategy for a given instance. While useful, offline tuning is not always feasible—e.g., when similar benchmarks suitable for tuning are unavailable. Moreover, offline tuning may require significant time and effort, and even when this overhead can be justified by an overall performance gain, dividing the solving procedure into offline and online phases makes the overall process more brittle and less automated. Motivated by these observations, we investigate the following question in this paper: *Can we customize the solving strategy for a given SMT*

problem based solely on the problem itself, without any offline strategy tuning?

Our key idea is to view a given SMT problem as a generator of relevant, easier subproblems and to use the generated subproblems to evaluate solving strategies. Existing formula-partitioning methods used in divide-and-conquer-based parallel SMT solving are natural candidates for the subproblem generation step, i.e., they can be used to partition a problem into subproblems from which we can sample for the purpose of tuning. The challenge with this approach, however, is that none of the subproblems are guaranteed to be sufficiently easy to solve. Consequently, tuning on such subproblems could require significant effort, possibly even more than would be required to solve the original problem itself.

To address this challenge, we propose a new subproblem generation technique for producing subproblems of bounded difficulty. Our key observation is that when an SMT solver backjumps during the execution of the CDCL(T) procedure, the subproblem corresponding to the set of decisions made so far at the time of the backjump is known to be solvable. Leveraging this observation, we design a procedure that either solves the full SMT problem or generates easy subproblems (measured by the number of conflicts required to solve the subproblem). Our approach produces subproblems that are significantly easier to solve than those generated by existing formula-partitioning algorithms.

Since the generated subproblems are used to evaluate and choose among candidate solving strategies, a natural follow-up question is: *Does a good solving strategy for the subproblems translate to a good strategy for the original problem?* We empirically demonstrate that the answer is positive for the subproblem-generation procedures we tested.

Building on this result, we propose an online per-instance strategy selection methodology: given an SMT problem, extract a set of easier subproblems, rank candidate solving strategies on those subproblems, and then tackle the original problem based on the ranking. On a selection of SMT-LIB benchmarks across 4 logics (QF_IDL, QF_LIA, QF_LRA, and QF_SLIA), we demonstrate that, when our method can successfully generate a ranking, it can reduce the total computation time over a parallel portfolio approach by an average of 27.8%.

To summarize, our contributions include:

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0020347. This work was also supported by DARPA under Agreement FA8750-24-9-1000 and by the Stanford Center for Automated Reasoning. Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

- a workflow for performing per-instance strategy selection for SMT completely online, without any offline tuning;
- a novel subproblem generation procedure that can be used to rank solving strategies more efficiently than existing formula-partitioning methods;
- the empirical discovery that strategy performance on subproblems translates to performance on the original problem for various subproblem-generation methods;
- an implementation of the proposed methodology; and
- an experimental evaluation demonstrating that the proposed method can reduce total computation time.

The rest of this paper is organized as follows. Section II and Section III cover preliminary information and related work, respectively. Section IV introduces our strategy selection methodology and the new subproblem generation procedure. Section V details our implementation, and Section VI presents our experimental evaluation. Finally, Section VII concludes the paper and discusses directions for future work.

II. PRELIMINARIES

Logical setting We assume the standard many-sorted first-order logic setting with the usual notions of signature, term, and interpretation. A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ -interpretations. For convenience, we assume a fixed background theory $\mathcal{T}$ with signature Σ including the Boolean sort `BOOL`. We further assume that all terms are Σ -terms, that entailment ($\models$) is entailment modulo $\mathcal{T}$, equivalence is equivalence modulo $\mathcal{T}$, and that interpretations are $\mathcal{T}$ -interpretations. A formula φ is a term of sort `BOOL` and is *satisfiable* (resp., *unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A formula whose negation is unsatisfiable is *valid*. The *satisfiability modulo theories* (SMT) problem is that of determining the satisfiability of an arbitrary formula.

CDCL($\mathcal{T}$)-based SMT solvers solve problems through the cooperation of a Boolean satisfiability (SAT) solver and one or more theory solvers [1]. In this framework, the SAT solver is responsible for constructing a truth assignment M , typically incrementally, that satisfies the Boolean abstraction of the input formula. At any point during the run, the subset of M consisting of decisions made by the SAT solver is called the *decision trail*. As M is built, each new assignment prompts a call to one or more theory solvers to check whether M remains consistent with the corresponding theories. If some theory solver discovers an inconsistency, it returns a *conflict clause* (and optionally new lemmas) to the SAT solver. A conflict clause is a disjunction of literals, valid in the theory, that is falsified under M , while a lemma is any other heuristically-chosen valid formula. Upon receiving a conflict clause, the SAT solver analyzes the conflict and performs a backtrack, removing part of the decision trail and learning the conflict clause to prevent encountering the same conflict again. This iterative process continues until one of two outcomes is achieved: either M is a complete satisfying assignment and no conflicts are detected by the theory solvers, meaning the

problem is satisfiable; or, an unrecoverable conflict is derived, and the problem is therefore unsatisfiable.

Strategy selection is the process of choosing a solving strategy for an SMT problem. A solving strategy is a tuple consisting of the solver and a set of options. In the evaluation, the solver is fixed, so we refer to the option sets as strategies. For clarity, we try to avoid using the word “strategy” outside of this context throughout the paper.

Subproblem generation procedures create a subproblem ϕ_i of the form $\phi \wedge s_i$, where ϕ is the original problem and s_i corresponds to a specific part of ϕ ’s search space. One common approach to creating subproblems is partitioning, such as cube-and-conquer [2] and scattering [3]. When partitioning, the goal is to produce N subproblems, each with difficulty $1/N$. Typically, the disjunction $\phi_1 \vee \dots \vee \phi_N$ is equisatisfiable with ϕ . In cube-and-conquer partitioning for SMT, N variables in the Boolean abstraction of the SMT problem are selected, and 2^N cubes (i.e., conjunctions of literals) are created. The scattering strategy produces disjoint partitions as follows. The first partitioning formula is a cube C_1 . The second uses a new cube, C_2 , to create the partitioning formula $\neg C_1 \wedge C_2$, and so on. The final partitioning formula is $\neg C_1 \wedge \dots \wedge \neg C_{N-1}$. We compare our new subproblem generation procedure to each of these approaches. While it is similar to the cube-and-conquer approach in that each s_i is a cube, our procedure does not attempt to evenly divide the problem.

III. RELATED WORK

Our approach is inspired and informed by several existing lines of work.

1) *Subproblem generation*: Previously, the main motivation for designing algorithms for generating subproblems from a given SMT problem has been to improve the effectiveness of divide-and-conquer-based SMT solving approaches [4], [5], [6]. In that context, the original problem is partitioned into many subproblems, each of which is solved in parallel. While we show that solver performance on the subproblems generated via existing partitioning algorithms does indeed predict the performance when solving the original problem, we elect not to use these algorithms, because the subproblems they generate are often too difficult. We develop instead a specialized subproblem generation procedure that efficiently generates subproblems that are easy to solve. Note that unlike prior work on subproblem generation, our new procedure does not *partition* the problem (i.e., the disjunction of the generated subproblems is not guaranteed to be equisatisfiable with the original problem).

2) *Per-instance strategy selection*: Traditional per-instance algorithm selection methods [7], [8], [9], [10] typically utilize machine learning to predict the most effective algorithm for a given problem. These methods assume the availability of a set of training problems, which is used to train an oracle that predicts the best solving strategy for a given problem based on its structural characteristics (e.g., a selected set of problem features used as the input to the oracle). We also

aim to customize the solving strategy for a given instance, but we do not assume the availability of training problems. Rather, we generate training problems from the given problem as needed. Our method is also distinct from prior approaches because it does not require distilling a set of features from a problem (i.e., inputs to the strategy-selecting oracle), which can be challenging.

3) *Online learning*: Our approach can be viewed as a form of online learning, which has been explored in various automated reasoning contexts. The MapleSAT solver [11], [12] was among the first to explore such a direction. For example, in MapleSAT, branching is formulated as a multi-armed bandit problem. The idea is to treat each variable as an arm and maintain its estimated reward throughout the solving. More broadly, algorithms with adaptive components are common in automated reasoning tools (e.g., dynamic local search techniques [13], branching heuristics [14], [15], and restart techniques [16]). In contrast, our method focuses on learning high-level solving strategies instead of low-level heuristics. There has also been work on choosing solving strategies online when solving a sequence of related problems [17], [18]. Our approach goes one step further by conducting online strategy selection for solving a single problem.

The most closely related recent work is on repurposing existing cubing algorithms to generate tuning data [19] for solving SAT and neural network verification problems. This paper explores a similar direction, but it is different in three important ways: (i) we study a new general-purpose SMT solving setting; (ii) instead of relying on existing cubing algorithms for subproblem generation, we design a new, specialized subproblem generation procedure; and (iii) instead of accelerating the solving of cubes in a cube-and-conquer procedure, our approach is aimed at boosting the performance when solving the original problem.

IV. SUBPROBLEM GENERATION FOR STRATEGY SELECTION

In this section, we describe our methodology for using subproblem generation for strategy selection. We start with a top-level overview of the approach (Algorithm 1). We then describe our new technique for subproblem generation (Algorithm 2). We finish the section with an explanation of how we solve subproblems and rank strategies based on the results of solving subproblems. In future sections, we will refer to subproblems generated using Algorithm 2 as *easy-cubes* or *ECubes* to abbreviate.

A. Overview

Algorithm 1 takes as input a formula to be checked for satisfiability. It then creates subproblems and uses them to rank solving strategies for that formula. The algorithm has several parameters, and we discuss specific ways of instantiating them below. We start by walking through the execution of Algorithm 1 step by step. We then discuss specific details and design decisions.

Algorithm 1 Generation of the ranked list of solving strategies for a given problem.

Input: ϕ , an SMT formula

Input: $stratList$, a list of solving strategies

Input: $N \geq 1$, the number of subproblems to solve

Output: $(status, rankedList)$

```

1:  $res, subprobs \leftarrow makeSubproblems(\phi, N)$ 
2: if  $res \in \{SAT, UNSAT\}$  then
3:   return  $(res, [])$ 
4: end if
5: if  $subprobs.size() < N$  then
6:   return  $(FAIL, [])$ 
7: end if
8:  $res, stratInfo \leftarrow solveSubprobs(subprobs, stratList)$ 
9: if  $res \in \{SAT, UNSAT\}$  then
10:  return  $(res, [])$ 
11: end if
12:  $rankedList \leftarrow makeRanking(stratInfo)$ 
13: return  $(RANKED, rankedList)$ 

```

Algorithm 1 is implemented outside an SMT solver. It takes as input an SMT formula, ϕ , a list of candidate solving strategies, $stratList$, and the number of subproblems to generate, N . On line 1, *makeSubproblems* is called to generate subproblems from ϕ . The implementation of *makeSubproblems* requires a subproblem generation procedure and optional code to, for example, increase diversity of subproblems. A new subproblem generation procedure is given in algorithm 2 and described in detail below, and diversification tactics are discussed in section V-B. Because it is possible for the problem to be solved during subproblem generation, *makeSubproblems* returns a tuple containing both a status and a list of subproblems. If the status is one of *SAT* or *UNSAT*, this means that ϕ was solved during subproblem generation. In this case, the status is returned with an empty list (lines 2-4). If fewer than N subproblems are generated by *makeSubproblems*, then the status *FAIL* is returned with an empty list (lines 5-7). Otherwise, *solveSubprobs* is called, which attempts to evaluate the candidate strategies in $stratList$ on the subproblems in $subprobs$. The specific implementation of *solveSubprobs* is discussed in Section V. Once again, *solveSubprobs* returns a tuple, because it is possible (depending, as we explain below, on how *makeSubproblems* and *solveSubprobs* are implemented) for the original problem to be solved during the call to *solveSubprobs*. If this happens, then once again, the status is returned with an empty list (lines 9-11). If not, $stratInfo$ is used in the call to *makeRanking* on line 12 to produce a ranked list of solving strategies according to a ranking heuristic, and this list is returned with the *RANKED* status.

Notice that there are three situations in which a ranked list cannot be generated: when the problem is solved during subproblem generation, when fewer than N subproblems are generated, and when the problem is solved during subproblem

solving (lines 3, 6, and 10, respectively). In the first and last cases, Algorithm 1 solves the problem, so producing a ranking is no longer necessary. However, in the case that not enough subproblems are generated, the method fails, producing neither a result nor a ranking. Each of the functions called in Algorithm 1 can be customized to use a specific subproblem generation procedure, different subproblem solving strategies, or a custom ranking heuristic. We discuss each of them in turn below.

Algorithm 2 Subproblem generation, embedded in the CDCL(T) framework.

Input: ϕ , an SMT formula

Input: *numSubprobs*, the number of desired subproblems

Output: (*status*, *subprobs*)

```

1: subprobsMade  $\leftarrow$  0
2: subprobs  $\leftarrow$  []
3: while True do
4:   conflict  $\leftarrow$  propagate()
5:   if conflict  $\neq$  {} then
6:     decisionLevel  $\leftarrow$  resolveConflict(conflict)
7:     if decisionLevel < 0 then
8:       return (UNSAT, [])
9:     end if
10:    if dumpCondition() then
11:      subprobs  $\leftarrow$   $\phi \wedge$  dumpDecisionTrail()
12:      subprobsMade  $\leftarrow$  subprobsMade + 1
13:      if subprobsMade == numSubprobs then
14:        return (UNKNOWN, subprobs)
15:      end if
16:    else if abortCondition() then
17:      return (UNKNOWN, subprobs)
18:    end if
19:    backtrack(decisionLevel)
20:  else
21:    if !makeDecision() then
22:      return (SAT, [])
23:    end if
24:  end if
25: end while

```

B. Subproblem Generation

The goal of subproblem generation is to produce a set of subproblems that are difficult enough to meaningfully distinguish among the different strategies, but easy enough to be solved relatively quickly, so as not to introduce too much overhead. A key contribution of this paper is a new algorithm for achieving this goal. Notice again that this is fundamentally different from the goal of partitioning for divide-and-conquer algorithms, discussed previously.

Algorithm 2 shows our new algorithm. Because it is implemented by instrumenting an SMT solver, we have included simplified pseudocode for a standard CDCL(T) implementation with our changes highlighted in blue. As before, we start by walking through the execution of Algorithm 2.

In addition to the SMT formula ϕ , Algorithm 2 takes *numSubprobs* as input, which specifies the desired number of subproblems that should be generated. On line 1, *subprobsMade* is initialized to 0 in order to start tracking the number of subproblems. The list of subproblems is similarly initialized to be empty on line 2. The main changes occur immediately before the call to *backtrack* on line 17. The inserted code is guarded by a call to *dumpCondition*() which is a callback function whose role is to check any conditions that must be fulfilled before generating a subproblem. If *dumpCondition*() returns *true* (e.g., the solver is sufficiently “warmed up”), then a new subproblem consisting of the conjunction of ϕ and the decisions in the decision trail is added to *subprobs*, and *subprobsMade* is incremented (lines 11-12). If the number of subproblems is equal to the target specified by *numSubprobs*, then the solver returns *UNKNOWN* (signaling that the subproblem generation did not solve the problem) and the list of generated subproblems (lines 13-14). Otherwise, if some *abortCondition*() holds (e.g., some time limit is reached), then *UNKNOWN* is returned together with whatever subproblems have been generated so far. We discuss specific instantiations of *dumpCondition*() and *abortCondition*() in the next section.

Notice that solving is unperturbed unless the requested number of subproblems is created or the abort condition holds. The solver can simply solve the problem as usual and return *SAT* or *UNSAT* (lines 8 and 22) if the conditions on lines 13 and 16 are never met. This behavior is important because it allows the instrumented SMT solver to potentially solve the original formula during subproblem generation. Its importance will become clearer in the discussion of implementation details. Briefly, though, the ability to solve the problem avoids wasting time tuning on subproblems for SMT formulas that are easy to solve. Conversely, it is also important to limit the amount of work done (via *abortCondition*()) during subproblem generation so as not to introduce too much overhead for more difficult problems.

The key insight of Algorithm 2 is that a subproblem can be generated from a discarded portion of the search space. At line 10, it is known that the current state of the decision trail results in a conflict. At the same time, the trail has not yet been backtracked, so it can be used to generate a subproblem. Intuitively, the generated subproblem should be easy to solve because we know it leads to a conflict. The subproblem should also require some nontrivial effort from the solver, since a conflict had to be produced.

It is important to note that while the generated subproblems are designed to be efficiently solvable, the actual runtime performance of the candidate solving strategies on those subproblems might vary significantly. In addition to the intrinsic differences in search patterns induced by different solving strategies, SMT solvers are known to exhibit high sensitivity to formula perturbations in some cases [20]. Adding the decision trail to the original problem may, therefore, result in unexpected runtimes for subproblem solving. Despite these caveats, we show in Section VI that, in practice, subproblems

generated using Algorithm 2 are frequently both efficient to solve and helpful for selecting good strategies.

Finally, we highlight a few features of Algorithm 2 that contribute to its utility for selecting solving strategies when incorporated into a tuning approach. First, we note that when algorithm 2 is used for *makeSubproblems* in algorithm 1, it can be beneficial to vary the options to provide a greater diversity of subproblems. Additionally, through *numSubprobs*, Algorithm 2 allows flexibility in the number of subproblems that are created. This flexibility can be used to adjust the execution time of subproblem generation to balance it with the other stages of the algorithm. *dumpCondition()* can also be used to help balance execution time, but more generally, it is the main way that this algorithm enables fine-grained control over subproblem generation. Depending on how *dumpCondition()* is defined, it can affect subproblem diversity, the time required to generate subproblems, and other characteristics of the subproblem generation process. We discuss various implementations of *dumpCondition()* in the next section. Lastly, notice that because subproblems are generated by decisions that led to a conflict, they have the additional characteristic that they are always unsatisfiable. Our evaluation shows that tuning on only these unsatisfiable subproblems can be effective. Intuitively, it also makes sense, as unsatisfiable problems require the solver to cover the entire search space, and if the solver can do that efficiently, it should also be able to solve satisfiable instances quickly.

C. Solving Subproblems

The *solveSubprobs* routine takes as input a list of subproblems and a list of strategies. Its goal is to capture information about the performance of the different strategies on the subproblems. There are different ways this can be done. For simplicity, we simply run all subproblems using all strategies using some given time limit. When using Algorithm 2 for subproblem generation, solutions to subproblems can never lead directly to solutions to the original problem because: (i) they cannot, by construction, be satisfiable, and (ii) they do not fully partition the problem, so even if they are all *UNSAT*, we can't conclude anything about the original problem. However, when the disjunction of the subproblems is equisatisfiable with the original formula, as with formula-partitioning procedures, it is possible to solve the original problem based on the results of the subproblems. In this case, if any subproblem returns *SAT*, we can immediately return with status *SAT*. Similarly, if all generated partitions are solved and found to be *UNSAT*, then we can immediately return with status *UNSAT*. When the status of the original problem cannot be deduced based on the result of solving its subproblems, we return the status *UNKNOWN* together with the performance data obtained by running all the strategies on all the subproblems.

D. Subproblem-based Strategy Ranking

Algorithm 1 is parameterized by the *makeRanking* method. We propose ranking solving strategies based on the number of subproblems solved and breaking ties using the total runtime

on the subproblems that were solved (a smaller total runtime is better). Note that ranking on time alone could result in solving strategies that solve fewer subproblems being placed higher in the ranked list. Since our priority is solving the original problem, we prioritize solving subproblems in our ranking.

V. IMPLEMENTATION DETAILS

We implemented the subproblem generation method (Algorithm 2) by extending CVC5's internal SAT solver. Our prototype of Algorithm 1 was implemented in Python and makes subprocess calls to CVC5 for subproblem generation (*makeSubproblems*) and solving (*solveSubprobs*). Each strategy in *stratList* is a set of command-line options for CVC5. In this section, we discuss specific design choices we made during the implementation.

A. The Dump and Abort Conditions

The *dumpCondition()* and *abortCondition()* methods in Algorithm 2 are configurable via command-line options in our implementation. *dumpCondition()* implements a delay before subproblem generation is enabled. The motivation is to ensure a sufficient amount of meaningful activity has occurred before producing subproblems. *abortCondition()* also relies on tracking time. In our evaluation, it returns true whenever it has been more than five seconds since the start of the current invocation of Algorithm 2.

B. Making Subproblems

We implement two versions of *makeSubproblems*. The first is based on Algorithm 2 and the other on standard formula-partitioning methods. In the first implementation, Algorithm 2 is invoked repeatedly to generate a small number of subproblems. Each call uses one of the strategies in *stratList* and requests $N/\text{stratList.size}()$ subproblems. Using the options from each of the strategies while generating subproblems helps generate diverse subproblems while mitigating the risk that subproblems are biased toward any single strategy. To further aid diversity, each call sets the time delay in *dumpCondition()* to a random time between 1.5 and 2.5 seconds. Importantly, only unique subproblems are used for tuning. If subproblem generation ever produces identical subproblems, the duplicate is discarded.

After this first round of generating subproblems, if we do not yet have N subproblems, we enter another loop. In this loop, we randomly pick a strategy and use it to generate $N/\text{stratList.size}()+1$ subproblems (we add one to increase the chances of getting new subproblems). As before, duplicates of any previously generated subproblem are removed. Whenever a strategy fails to produce any new subproblems, it is no longer eligible to be picked. The loop continues until N subproblems are generated or until all strategies have been deemed ineligible.

In the second mode where partitioning is used, we use the built-in partitioning capabilities of CVC5. Depending on the run and the partitioning procedure used, the number of subproblems generated may not match N exactly. If fewer

TABLE I: Options used for each logic.

Logic	(Alias) Options
QF_LRA, QF_RDL	(A) --miplib-trick --miplib-trick-subs=4 --use-approx --use-soi --lemmas-on-replay-failure --replay-early-close-depth=4 --replay-lemma-reject-cut=128 --replay-reject-cut=512 --unconstrained-simp
	(B) --no-restrict-pivots --use-soi --new-prop --unconstrained-simp
	(C) defaults
QF_LIA, QF_IDL	(D) --miplib-trick --miplib-trick-subs=4 --use-approx --lemmas-on-replay-failure --replay-early-close-depth=4 --replay-lemma-reject-cut=128 --replay-reject-cut=512 --unconstrained-simp --pb-rewrites --ite-simp --simp-ite-compress --no-use-soi
	(E) --miplib-trick --miplib-trick-subs=16 --use-approx --lemmas-on-replay-failure --replay-early-close-depth=4 --replay-lemma-reject-cut=16 --replay-reject-cut=64 --unconstrained-simp --pb-rewrites --ite-simp --simp-ite-compress --use-soi
	(C) defaults
	(C) defaults
QF_SLIA	(F) --strings-exp --no-jh-rlv-order
	(G) --strings-exp --strings-fmf --no-jh-rlv-order

than N are generated, we just exit, and Algorithm 1 fails. If more than N subproblems are generated, then N of them are randomly selected. For partitioning runs, CVC5 is limited to 60 seconds and is called using the default options.

C. Solving Subproblems and Ranking

Our implementation of *solveSubprobs* attempts to solve each subproblem in *subprobs* with each of the strategies in *stratList* using CVC5. A solving timeout of 5 seconds is enforced for each attempt at solving a subproblem. The returned *stratInfo* is a dictionary mapping the solving strategies to a tuple describing how many subproblems were solved using each solving strategy and the cumulative runtime for that solving strategy over the subproblems. Note that having more strategies in *stratList* may increase the odds of finding the best solving strategy, but it comes at the cost of running each subproblem an additional time. Rankings are calculated in *makeRanking* based on the ranking heuristic described in Section IV-D.

VI. EXPERIMENTAL EVALUATION

In this section, we report on experiments designed to evaluate our approach. The main questions we investigate are:

- How does *Ecubes* compare to subproblem generation methods based on partitioning? [It consistently ranks more problems]
- Does solving strategies' performance on subproblems predict their performance on the original problem? [Yes for all tested subproblem generation methods]
- Can the proposed online strategy selection method reduce computational time? [In many cases]

A. Experimental Setup

We consider quantifier-free benchmarks from the following logics in the SMT-LIB benchmark library: QF_LRA, QF_LIA, QF_RDL, QF_IDL, and QF_SLIA. The first four were selected

because they have been studied in previous work on partitioning [4], and specific solving strategies for those benchmarks have been published in the context of portfolio solving [21]. We added QF_SLIA because efficient string solving has been of particular practical interest recently [22].

For candidate solving strategies, we start with expert-selected options for each logic found in CVC5's competition script. These options were expanded by varying the decision heuristic and including default options, which was an effective portfolio technique for QF_LRA, QF_LIA, QF_RDL, and QF_IDL in [21]. We extend the method of varying the decision heuristic for expert-selected options to the QF_SLIA logic. Concretely, we consider the Cartesian product of the option sets shown in Table I and CVC5's three decision heuristics: justification, stop-only, and internal. In total, there are six solving strategies (option sets) for QF_SLIA, and nine for the other benchmark sets. As shown in Figure 1, each of these solving strategies is effective for many problems in each logic, with each strategy being the best single strategy for some portion of the benchmarks.

All experiments were run on a cluster with 48 nodes running Ubuntu 20.04 LTS, each with one AMD Ryzen 9 7950X CPU with 16 cores and just under 128 GB of RAM (127940MB).

We start by walking through a detailed analysis of our methods on the QF_LRA benchmarks (in the next three subsections); we then broaden our scope to include other logics. We focus first on a single logic to keep the analysis simple and understandable, and we choose these benchmarks because the majority of prior work on subproblem generation includes and often focuses on QF_LRA benchmarks.

B. Comparison of Techniques for Subproblem Selection

We compare three subproblem generation methods: The first method is the one based on Algorithm 2 and detailed in Section V-B. Recall that we call this method *easy-cubes*, or *ECubes* for short. The others are partitioning algorithms implemented in CVC5 [4]. One is a cube-and-conquer based approach that builds cubes using literals from the decision trail. We call this the *decision cube* method, or *Dcube* for short. The last method is based on scattering using literals from the decision trail. We call this *Dscatter*.

The metric used for comparison is *rankability*. A problem is rankable for a particular subproblem generation method if running Algorithm 1 using that method does not return *FAIL* as a status and if the top strategy solves at least three subproblems (solving fewer than three suggests there is not enough data to meaningfully distinguish the strategies).

SMT-LIB contains 1753 QF_LRA problems. Of these, 1728 can be solved by at least one of the solving strategies. And of these, 517 are not trivially solved during subproblem generation (using *Ecubes*). We thus compare our subproblem generation methods on these remaining 517 benchmarks. We ran Algorithm 1 on each benchmark using several variations of our subproblem generation methods using an overall timeout of 1200s for each run.

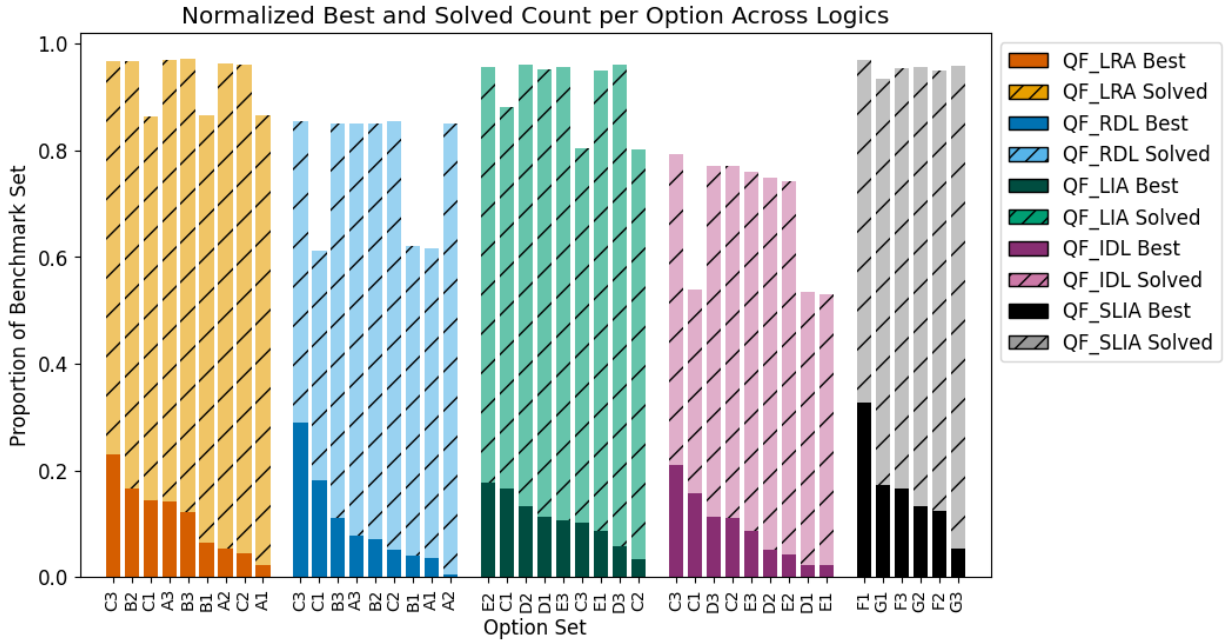


Fig. 1: This graph shows the proportion of problems that each option set can solve in each logic and additionally highlights the proportion of problems for which each option set is the best. Option sets are of the form XN where X is the options alias in Table I and N corresponds to a decision heuristic: justification, stop-only, and internal are aliased as 1, 2, and 3, respectively.

The results are shown in Figure 2. The x-axis shows each subproblem generation procedure we tested. The three main approaches are parameterized by the number of subproblems generated (the first number in parentheses) and the number of subproblems used (the second number). Recall that for partitioning methods, we may generate more subproblems than we use. In this case, we randomly sample from the generated subproblems (results using random sampling are highlighted in the figure as a reminder). For example, Ecube (18) 18 generates 18 easy-cubes and uses all of them, whereas Dcube (256) 27, generates 256 decision cubes and randomly selects 27 for tuning. The y-axis shows the number of problems successfully ranked. The Ecube variations result in the largest number of successful rankings, while Dscatter variants produce the fewest successful rankings. The results show that our new subproblem generation algorithm based on Algorithm 2 is the most effective for ranking our QF_LRA strategies on these benchmarks. Notice that the partition-based techniques struggle because the produced subproblems are too difficult, not because there are too few partitions being generated. For example, Dcube (256) 18 ranks more problems than Dcube (16) 16, even though creating 256 decision-cube partitions is more likely to fail than creating 16.

We next evaluate the difficulty of the subproblems generated by different methods. Recall that one goal is to produce relatively easy subproblems so that tuning does not take too long. Figure 3 visualizes the time required to solve all of the subproblems generated by each of the top four methods from Figure 2. Recall that we solve each subproblem with all 9 solving strategies, each with a 5s time limit. The horizontal

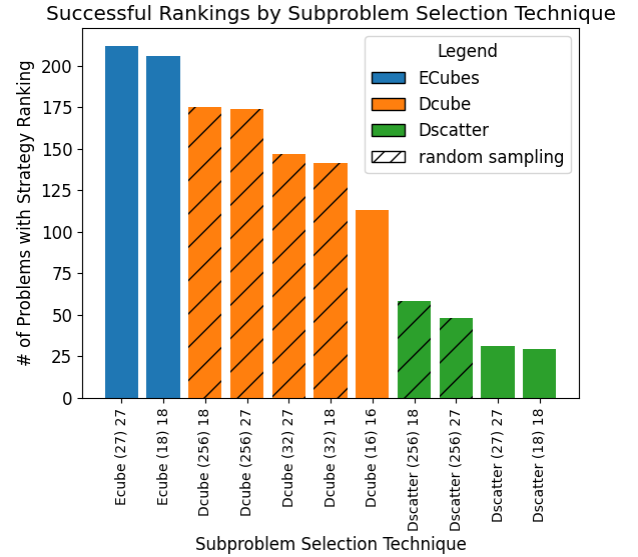


Fig. 2: Bar plot of the number of QF_LRA problems for which a strategy ranking can be produced using various subproblem selection techniques.

line in the boxplot is the median, the box itself represents the values in the lower to upper quartiles, and the whiskers extend to show the full range of the data. Figure 3 shows that the typical time to solve all subproblems is much lower when the subproblems were generated using the easy-cubes procedure than when they were generated using decision-cube

TABLE II: Comparison of the number of problems solved by top-ranked strategies to the expected number of problems solved by selecting a random solving strategy.

Technique	Common		All Ranked	
	Top Rank Solved	Random Solved	Top Rank Solved	Random Solved
Ecube (18) 18	119/133	103.3/133	181/206	161.2/206
Dcube (256) 18	118/133	103.3/133	152/175	135.7/175
Ecube (27) 27	119/133	103.3/133	184/212	165.9/212
Dcube (256) 27	117/133	103.3/133	143/174	135.9/174

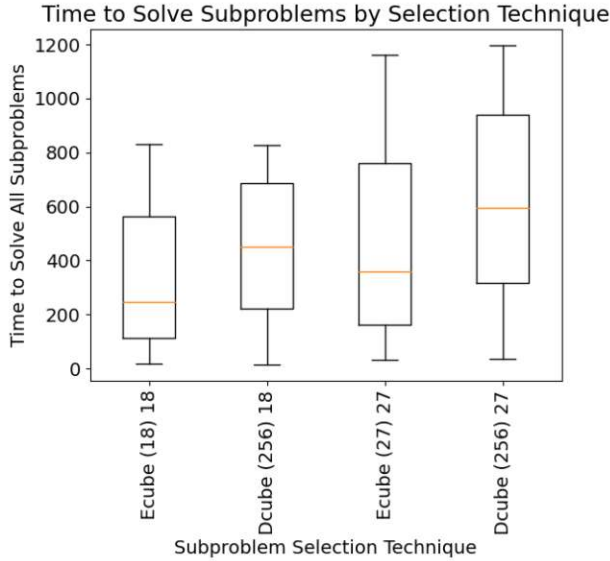


Fig. 3: Box plot showing the time required to solve all subproblems produced with a given subproblem selection technique.

partitioning, with the median of Dcube (256) 18 being almost double that of Ecube (18) 18.

C. Evaluation of the predictive power of rankings

So far, we have seen that, for QF_LRA, the Ecube method not only produces rankings more often, but also does so faster. However, it remains to be seen whether these rankings are useful for predicting a good strategy for the original problem. We evaluate this next. Concretely, we evaluate how often the top-ranked solving strategy can solve the original problem.

Table II shows the number of instances that are solved when using the top-ranked strategy for each subproblem selection technique. It also shows the expected number of instances solved by choosing a random strategy. “Common” reports results on the 133 benchmarks ranked by all four methods, whereas “All Ranked” reports results on all benchmarks that each method was able to rank. Regardless of how the subproblems are generated, the ranking successfully selects a good solving strategy with substantially higher accuracy than a random choice. This result shows that tuning on generated subproblems is a promising technique for strategy selection. Since each subproblem generation technique results

in a similar number of solved benchmarks, we conclude that the efficiency gain of Ecube (as shown in Section VI-B) does not come at the cost of reduced predictive power.

D. Comparison of online tuning with parallel portfolio

We now determine whether the per-instance strategy selection method could be used to improve solving efficiency in practice. We consider the following approach: first tune using Algorithm 1 and then iteratively attempt to solve the problem using the ranked solving strategies in order with a 20-minute per-strategy timeout. We compare the total computational time of this approach with that of a parallel portfolio solving method, where all strategies are attempted simultaneously (with a 20 minute timeout). The total compute time of our approach is computed as the sum of the time to produce subproblems, the time to solve each of the subproblems with all candidate solving strategies, and the time to try each strategy in the ranked list until the problem is solved. This means assigning a higher rank to a solving strategy that will not solve the problem is penalized because 1200 seconds will be added to the total time for each failed attempt. Note that each of the 6-9 option sets for each logic is the best strategy for some problems, as shown in Figure 1. While including ineffective strategies would clearly reduce the performance of the parallel portfolio, doing so could also sabotage our strategy: every option is considered during tuning, and including ineffective strategies could introduce noise into the ranking heuristic. We use the Ecube (18) 18 subproblem selection technique because it achieves a good balance between ranking many problems effectively and minimizing tuning time, as seen in prior subsections.

Given a set of problems and limited computational resources, online per-instance tuning and the parallel portfolio approach represent two distinct philosophies for utilizing those resources to tackle the problems. A parallel portfolio uses all available workers to attempt the same problem in parallel using different strategies with the hope that one strategy will solve it quickly. In contrast, online tuning solves a given problem with just one strategy at a time, but computational resources are spent on solving subproblems to choosing the strategy order.

We select benchmarks that (i) take at least 90 seconds to solve with the best strategy; (ii) are successfully solved by at least one strategy; and (iii) are successfully ranked using the Ecube (18) 18 method. We focus on harder benchmarks because our method is intended for solving problems where spending the resources required for tuning is justifiable. Note that easy benchmarks are likely to be solved during subproblem generation, so it makes sense to focus on harder problems.

Table III reports the total compute time of the two configurations across benchmark families, with the number of selected instances per family shown in the first column followed by, in parentheses, the number excluded due to each of (i), (ii), and (iii) above, respectively. Even though only three benchmarks remain in QF_RDL after the selection procedure, we include it for transparency.

TABLE III: Comparison of the total compute time required by a parallel portfolio and the Ecube (18) 18 tuning approach on selected problems from different benchmark sets. The aggregate improvement is reported with the per-instance median improvement in parentheses.

Logic Instances (Excluded)	Tactic	CPU Time (s)	Improvement (Median)
QF_IDL 73 (1745/463/247)	Parallel Portfolio	186123	32.5%
	Ecube (18) 18	125609	(52.4%)
QF_SLIA 51 (81661/2281/402)	Parallel Portfolio	87451	9.8%
	Ecube (18) 18	78845	(25.0%)
QF_LRA 23 (1654/25/51)	Parallel Portfolio	65248	50.2%
	Ecube (18) 18	32470	(77.2%)
QF_LIA 15 (12823/334/134)	Parallel Portfolio	47951	18.8%
	Ecube (18) 18	38934	(36.6%)
QF_RDL 3 (209/36/7)	Parallel Portfolio	9337	57.3%
	Ecube (18) 18	3987	(1.5%)

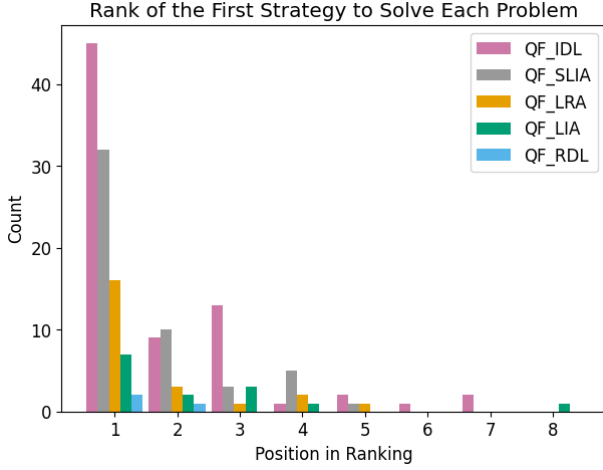


Fig. 4: Histogram of the rank of the first strategy to solve the problem using Ecube (18) 18 for the benchmarks reported in Table III.

Across all considered benchmark sets, our method results in an overall improvement in total compute time, ranging from 9.8% for QF_SLIA to 50.2% for QF_LRA. The per-instance median improvement demonstrates that for the majority of instances, total compute time is substantially reduced using our approach. Figure 4 shows the rank of the first strategy that solved the problem for each logic. The majority are solved by the top ranked strategy, as expected from the results in the previous subsection. Figure 5 shows the per-instance total compute time for the QF_LRA benchmarks. The scatter plot suggests that for these benchmarks, many benchmarks greatly reduce total compute time, while only a few use a modest amount of additional compute.

As shown in Table II, there are cases where our ranking method fails to put a successful strategy at the top, despite its overall effectiveness. Further analysis reveals that for those cases, the performance of the top-ranked strategy during tuning typically does not differ significantly from those of the other

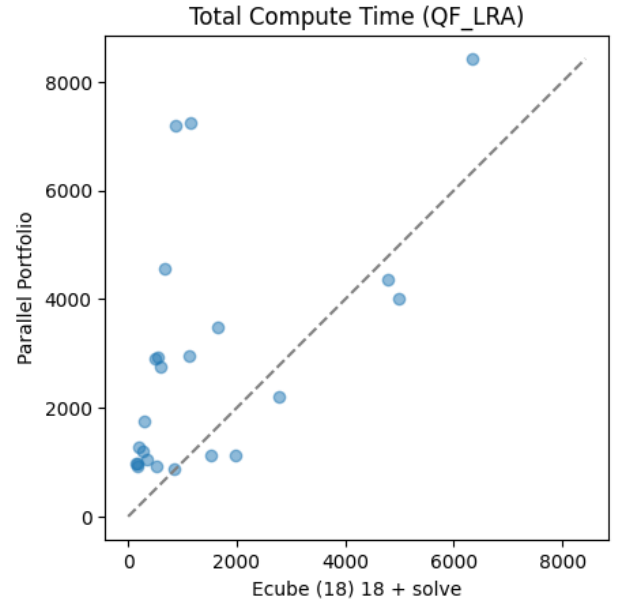


Fig. 5: Scatter plot of the total compute time of Ecube (18) 18 versus parallel portfolio.

TABLE IV: Comparison of the number of problems solved by top-ranked strategies to the expected number of problems solved by selecting a random solving strategy, when using stricter ranking criteria.

Technique	Common		All Ranked	
	Top Rank Solved	Random Solved	Top Rank Solved	Random Solved
Ecube (18) 18	60/62	49.6/62	136/149	117.8/149
Dcube (256) 18	59/62	49.6/62	136/144	112.1/144
Ecube (27) 27	60/62	49.6/62	144/166	130.4/166
Dcube (256) 27	58/62	49.6/62	126/143	111.8/143

strategies. Indeed, when we filter out benchmarks for which all strategies solve the same number of subproblems, the ability of our method to predict a strategy that will solve the original problem jumps—attaining an accuracy of over 90%, as shown in Table IV. Future work could consider how to best leverage this observation. For example, it could be beneficial to generate additional subproblems when the current set fails to produce a meaningful difference in scores for the candidate solving strategies.

VII. CONCLUSION AND FUTURE WORK

In this paper, we demonstrated that strategy performance on a given SMT formula can be predicted from strategy performance on its easier subproblems. We showed that the correlation holds for existing subproblem generation methods—formula partitioning methods used for divide-and-conquer-styled parallel solving—as well as for a novel subproblem generation method. The new method can produce a ranking of solving strategies more efficiently than existing methods without sacrificing the quality of the ranking. We then proposed a

solving methodology that first performs per-instance strategy selection online and then solves the instance with the ranked strategies. We showed that using this approach can reduce total compute time when compared to a portfolio approach that directly tries all strategies in parallel. Our paper is the first to demonstrate the feasibility of moving meta-algorithmic design *completely* online for SMT solving.

While our new subproblem generation method outperforms existing partitioning-based methods in our setting, there are still many challenging instances for which subproblems cannot be generated, as the complexity of the problem stems from theory reasoning rather than from the Boolean structure, or for which rankings cannot be produced because not enough subproblems are solved. This suggests it could be interesting to develop theory-specific subproblem generation methods or find ways to adjust the difficulty of subproblems by adding additional lemmas. Other future directions include 1) selecting among multiple solvers; 2) exploring a much larger pool of strategies using stochastic optimization techniques; and 3) validating the efficacy of the proposed workflow for other SMT solvers and theories.

REFERENCES

- [1] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to dpll(t),” *J. ACM*, vol. 53, no. 6, p. 937–977, nov 2006.
- [2] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, “Cube and conquer: Guiding CDCL SAT solvers by lookaheads,” in *Hardware and Software: Verification and Testing*, K. Eder, J. Lourenço, and O. Shehory, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 50–65.
- [3] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, “A distribution method for solving sat in grids,” in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–435.
- [4] A. Wilson, A. Noetzi, A. Reynolds, B. Cook, C. Tinelli, and C. W. Barrett, “Partitioning strategies for distributed smt solving,” in *FMCAD*, 2023, pp. 199–208.
- [5] A. E. Hyvärinen, M. Marescotti, and N. Sharygina, “Search-space partitioning for parallelizing smt solvers,” in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24–27, 2015, Proceedings 18*. Springer, 2015, pp. 369–386.
- [6] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, “Lookahead in partitioning SMT,” in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 271–279.
- [7] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Satzilla: portfolio-based algorithm selection for sat,” *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.
- [8] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, “Machsmt: A machine learning-based algorithm selector for smt solvers,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 303–325.
- [9] L. Xu, H. Hoos, and K. Leyton-Brown, “Hydra: Automatically configuring algorithms for portfolio-based selection,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010, pp. 210–216.
- [10] R. Singh, J. P. Near, V. Ganesh, and M. Rinard, “Avatarsat: An auto-tuning boolean sat solver,” Technical Report MIT-CSAIL-TR-2009-039. Massachusetts Institute of Technology, Tech. Rep., 2009.
- [11] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning rate based branching heuristic for SAT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140. [Online]. Available: https://doi.org/10.1007/978-3-319-40970-2_9
- [12] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh, “Machine learning-based restart policy for cdcl sat solvers,” in *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*. Springer, 2018, pp. 94–110.
- [13] C. M. Li, W. Wei, and H. Zhang, “Combining adaptive noise and look-ahead in local search for sat,” in *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28–31, 2007. Proceedings 10*. Springer, 2007, pp. 121–133.
- [14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 530–535.
- [15] M. S. Cherif, D. Habet, and C. Terrioux, “Combining vsids and chb using restarts in sat,” in *27th International Conference on Principles and Practice of Constraint Programming*, 2021.
- [16] A. Biere, “Adaptive restart strategies for conflict driven sat solvers,” in *Theory and Applications of Satisfiability Testing–SAT 2008: 11th International Conference, SAT 2008, Guangzhou, China, May 12–15, 2008. Proceedings 11*. Springer, 2008, pp. 28–33.
- [17] N. Pimpalkhare, F. Mora, E. Polgreen, and S. A. Seshia, “Medleysolver: online smt algorithm selection,” in *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5–9, 2021, Proceedings 24*. Springer, 2021, pp. 453–470.
- [18] H. Wu, C. Hahn, F. Lonsing, M. Mann, R. Ramanujan, and C. Barrett, “Lightweight online learning for sets of related problems in automated reasoning,” in *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2023, pp. 1–11.
- [19] H. Wu, C. Barrett, and N. Narodytska, “Cubing for tuning,” 2025. [Online]. Available: <https://arxiv.org/abs/2504.19039>
- [20] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, “Mariposa: Measuring smt instability in automated program verification,” in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, 2023, pp. 178–188.
- [21] C. Barrett, P.-W. Chen, B. Cook, B. Dutertre, R. B. Jones, N. Le, A. Reynolds, K. Sheth, C. Stephens, and M. W. Whalen, “Smt-d: New strategies for portfolio-based smt solving,” in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 1–10.
- [22] 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)*. IEEE, 2018, pp. 1–9.

Solving Set Constraints with Comprehensions and Bounded Quantifiers

Mudathir Mohamed¹, Nick Feng², Andrew Reynolds¹,
Cesare Tinelli¹, Clark Barrett³ and Marsha Chechik²

¹The University of Iowa ²University of Toronto ³Stanford University

Abstract—Many problems in real-world applications can be expressed easily as quantified formulas in SMT. Unfortunately, reasoning about quantified formulas in SMT is notoriously difficult. While different strategies and quantifier instantiation techniques have been developed to tackle this challenge, SMT solvers still struggle with quantified formulas generated by some applications. We discuss the use of *set-bounded quantifiers*, quantifiers whose variable ranges over a finite, but possibly unbounded, set. Such quantifiers can be eliminated using the quantifier-free fragment of the theory of finite relations extended with a filter operator, which provides a form of restricted set comprehension. We show that this reduction to filter constraints outperforms other quantification techniques in satisfiable problems generated by the SLEEC tool, and is very competitive on unsatisfiable benchmarks with LEGOS, a specialized solver for SLEEC. We also identify a decidable class of constraints with restricted applications of the filter operator, while showing that unrestricted applications lead to undecidability.

I. INTRODUCTION

Problems from many real applications can be encoded naturally in the language of finite sets and relations. Concrete examples include software design specifications [16], ontologies [18], database queries [21], normative requirements [13], and authorization policies [10]. Some SMT solvers support the theory of finite sets and relations [1], [18]. Possible encodings in SMT from the applications listed above involve the use of quantifiers with variables that range over elements of some sets with possibly unbounded but finite cardinality. A typical approach is to use standard quantifiers and rely on general quantifier instantiation techniques implemented in SMT solvers. However, solver performance is generally poor in these cases because of the inherent difficulty of reasoning modulo theories about quantified formulas.

The *cvc5* solver supports a theory of finite sets and relations that was recently extended with a second-order filter operator [21]. The filter operator implements a form of set restricted comprehension that constructs a subset from a given set using a given predicate. The extension makes it now possible to express set-bounded quantification using only quantifier-free formulas. This presents an opportunity to encode problems that involve bounded quantification as quantifier-free problems, avoiding the challenges of reasoning with quantifiers.

In this paper, we discuss how to encode set-bounded quantifiers in a subtheory of *cvc5*'s theory of finite sets and relations. We also present very encouraging initial experimental results that evaluate this encoding on a set of benchmarks from real-world problems. Finally, we discuss the theoretical question

of whether there is a reasonably expressive logical fragment with set-bounded quantification with a decidable satisfiability problem. We answer the question positively, but also show that restrictions on the use of the filter operator are necessary for decidability.

This work makes the following **contributions**:

- 1) A decision procedure for the satisfiability of a large class of quantifier-free formulas for a theory of finite sets and relations extended with the Cartesian product operator and a filter operator.
- 2) A high-level proof that the unrestricted use of the filter operator compromises our decidability result.
- 3) An approach for expressing set-bounded quantifiers in terms of quantifier-free formulas containing applications of the filter operator.
- 4) Initial empirical results over a set of benchmarks from a real-world application showing that an encoding to SMT using our approach leads to better solver performance compared to encodings using standard quantifiers.

A. Related Work

Multi-Level Syllogistic (MLS) theory can be taken as the core fragment for unsorted set theory [14]. It contains the basic operators $\sqcup, \sqcap, \setminus, \sqsubseteq, \approx, \in$ in addition to logical connectives $\vee, \wedge, \neg$. Its syntax and semantics consider only sets, with no scalar values, as the latter can be encoded as (nested) sets. A decision procedure based on singleton models¹ was given by Ferro et al. [14]. The same paper proved the decidability of an MLS extension with map variables² and operators that return as sets their domains and ranges.

Bansal et al. [1] presented an efficient decision procedure for a natural variant of MLS with a set cardinality operator in the context of many-sorted logic. Meng et al. [18] extended that theory with operators for finite relations, expressed as finite sets of tuples, that include cross product, relational join, transpose, and transitive closure. They proved the decidability of a small restricted fragment that only accepts binary relations along with relational join and transpose operators. Recently, Mohamed et al. [21] extended that theory further with a second-order filter operator, σ , and second order map operator, π , to support reasoning about SQL queries. It can be shown that MLS extended with a set map operator π is decidable

¹A singleton model is one that assigns each set variable either $\emptyset$ or $\{\emptyset\}$.

²Map variables are binary relations.

when its function argument is uninterpreted. This can be achieved by a reduction to the MLS extension with map variables in [14]. If interpreted functions are allowed in map terms, the fragment is undecidable in general, similarly to the undecidability result we present in Section III-D.

Set-bounded quantifiers have been studied extensively for unsorted theories of finite sets. Cantone et al. [5] showed the undecidability of alternating set quantifiers $(\forall\exists)_0$. Some restricted fragments that are decidable are mentioned in Cantone [6]. $\text{MLSSF}^\forall$ is an example of a decidable fragment that extends MLS with uninterpreted unary functions, singletons, and positive universal bounded quantifiers only with some restrictions on their body expressions. $\text{MLSSF}^\forall$ is close in spirit to the decidable fragment we present in Section III for our many-sorted theory of sets, albeit the latter allows existential quantifiers in restricted cases. Cristiá et al. [9] discussed decidable fragments for “restricted” quantifiers (our set-bounded quantifiers): $\exists, \forall, \exists\forall$, and $\forall\exists$ with no cycles in the “domain graph.” Our calculus and implementation in *cvc5* decides the satisfiability of formulas in the first three fragments $\exists, \forall, \exists\forall$ but not in $\forall\exists$. Currently, it accepts $\forall\exists$ formulas but without termination guarantees. That said, our experimental evaluation does include $\forall\exists$ problems with alternating quantifiers, and our approach works reasonably well with those problems. We also note that our calculus supports the cross product operator, which is not mentioned in [9].

Deciding MLS extended with Cartesian product is reported as an open problem by [4], [6]. Cantone and Ursino [7] proved that MLS with an unordered Cartesian product operator but without the set membership operator is decidable in unsorted sets. We prove (see Corollary 1) that MLS with membership and Cartesian product operators is decidable in a sorted setting, where all elements of a set have the same sort.

II. FORMAL PRELIMINARIES

We define our theories and our calculi in the context of many-sorted logic with equality and polymorphic sorts and functions. We assume the reader is familiar with the following notions from that logic: signature, term, formula, free variable, interpretation, and satisfiability in an interpretation. Let Σ be a many-sorted signature. We will denote sort parameters in polymorphic sorts with α and β , and monomorphic sorts with τ . We will use $\approx$ as the (infix) logical symbol for equality — which has polymorphic rank $\alpha \times \alpha$ and is always interpreted as the identity relation over α . We assume all signatures Σ contain the Boolean sort Bool , always interpreted as the binary set $\{\text{true}, \text{false}\}$, and two Boolean constant symbols, $\top$ and $\perp$, for *true* and *false*. A Σ -term is a well-sorted term all of whose function symbols are from Σ . A Σ -formula is defined analogously and is constructed with the usual Boolean connectives and quantifiers out of atomic formulas. Without loss of generality, we assume $\approx$ is the only predicate symbol in Σ , as all other predicates can be modeled as functions with return sort Bool . If p is a Σ -term of sort Bool , we consider it as a predicate by treating it as a shorthand for $p \approx \top$. We write $s \not\approx t$ as an abbreviation for $\neg s \approx t$. If φ is a Σ -formula

and $\mathcal{I}$ is a Σ -interpretation, we write $\mathcal{I} \models \varphi$ if $\mathcal{I}$ satisfies φ . If t is a term, we denote by $\mathcal{I}(t)$ the value of t in $\mathcal{I}$. A *theory* is a pair $\mathbb{T} = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ -interpretations, the *models* of $\mathbb{T}$, that is closed under variable reassignment (i.e., every Σ -interpretation that differs from one in $\mathbf{I}$ only in how it interprets the variables is also in $\mathbf{I}$).

A Σ -formula φ is *satisfiable* (resp., *unsatisfiable*) in $\mathbb{T}$ if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. Two Σ -formulas φ_1 and φ_2 are *equisatisfiable in $\mathbb{T}$* if for every model $\mathcal{A}$ of $\mathbb{T}$ that satisfies one of the two formulas, there is a model of $\mathbb{T}$ that satisfies the other and differs from $\mathcal{A}$ at most in how it interprets the free variables not shared by φ_1 and φ_2 .

A. A Theory of Finite Relations

We define a many-sorted theory $\mathbb{T}_{\text{Rel}}$ of finite relations. Its signature Σ_{Rel} is given in Table I. We use α and β , possibly with subscripts, as sort parameters in polymorphic sorts. Additionally, $\mathbb{T}_{\text{Rel}}$ has three classes of sorts, with a corresponding polymorphic sort constructor: predicate sorts, tuple sorts, and set sorts. *Predicate sorts* are monomorphic instances of $\alpha_1 \times \dots \times \alpha_k \rightarrow \text{Bool}$ for all $k \geq 0$. *Tuple sorts* are constructed by the variadic constructor *Tuple* which takes zero or more sort arguments. For each $k \geq 0$, $\text{Tuple}(\tau_1, \dots, \tau_k)$ denotes the set of tuples of size k with elements of sort $\tau_1, \dots, \tau_k$, respectively. *Set sorts* are monomorphic instances $\text{Set}(\tau)$ of $\text{Set}(\alpha)$. The sort $\text{Set}(\tau)$ denotes the set of all *finite* sets of elements of the domain denoted by sort τ . We model relations as sets of tuples and write $\text{Rel}(\tau_0, \dots, \tau_k)$ as a shorthand for the sort $\text{Set}(\text{Tuple}(\tau_0, \dots, \tau_k))$.

Signature Σ_{Rel} , summarized in Table I, contains a subset of the set symbols from Mohamed et al. [21]. The semantics of the various set operators is the expected one. The operator $\in$ denotes the set membership relation and $\subseteq$ the set inclusion relation. Expressions $[], [e], s \sqcup t, s \sqcap t, s \setminus t, s \times t$ denote the empty set, the singleton set containing e and the union, intersection, difference, and flat Cartesian product of sets s and t , respectively. The filter operator σ is a second-order operator taking a predicate as its first argument. Terms $\sigma(p, s)$ denote the set consisting of the elements of set s that satisfy predicate p . We extend the language to allow lambda abstractions and their applications, and we allow the argument p of $\sigma(p, s)$ to be either a (second-order) variable or a lambda abstraction, both of rank $\alpha \rightarrow \text{Bool}$ if s has sort $\text{Set}(\alpha)$. For increased readability, we write $\sigma(p, s)$ as the set comprehension term $[x \mid x \in s \wedge p(x)]$.³

The signature contains also the set quantifiers *set.all* and *set.some*, which we discuss in detail in Section IV.

In practice, we are not interested in theory $\mathbb{T}_{\text{Rel}}$ in isolation but in combination with some theory $\mathbb{T}_{\text{El}}$ of set/relation elements (e.g., the theory of integers, strings, and so on). For the rest of the paper then, we fix an extension $\mathbb{T}'_{\text{Rel}}$ of $\mathbb{T}_{\text{Rel}}$ with signature Σ'_{Rel} . The extended theory incorporates a (possibly combined) theory of elements whose signature Σ_{El}

³We reserve $\{-\}, \in, \cup$, and so on to denote sets and set operations at the meta level.

TABLE I
SIGNATURE Σ_{Rel} FOR THE THEORY OF RELATIONS.

Symbol	Signature	SMT-LIB Syntax
$[]$	$\text{Set}(\alpha)$	<code>set.empty</code>
$[-]$	$\alpha \rightarrow \text{Set}(\alpha)$	<code>set.singleton</code>
$\sqcup$	$\text{Set}(\alpha) \times \text{Set}(\alpha) \rightarrow \text{Set}(\alpha)$	<code>set.union</code>
$\sqcap$	$\text{Set}(\alpha) \times \text{Set}(\alpha) \rightarrow \text{Set}(\alpha)$	<code>set.inter</code>
$\setminus$	$\text{Set}(\alpha) \times \text{Set}(\alpha) \rightarrow \text{Set}(\alpha)$	<code>set.minus</code>
$\in$	$\alpha \times \text{Set}(\alpha) \rightarrow \text{Bool}$	<code>set.member</code>
$\subseteq$	$\text{Set}(\alpha) \times \text{Set}(\alpha) \rightarrow \text{Bool}$	<code>set.subset</code>
σ	$(\alpha \rightarrow \text{Bool}) \times \text{Set}(\alpha) \rightarrow \text{Set}(\alpha)$	<code>set.filter</code>
<code>set.all</code>	$(\alpha \rightarrow \text{Bool}) \times \text{Set}(\alpha) \rightarrow \text{Bool}$	<code>set.all</code>
<code>set.some</code>	$(\alpha \rightarrow \text{Bool}) \times \text{Set}(\alpha) \rightarrow \text{Bool}$	<code>set.some</code>
$\langle \dots \rangle$	$\alpha_0 \times \dots \times \alpha_k \rightarrow \text{Tuple}(\alpha_0, \dots, \alpha_k)$	<code>tuple</code>
$\times$	$\text{Rel}(\alpha) \times \text{Rel}(\beta) \rightarrow \text{Rel}(\alpha, \beta)^1$	<code>rel.product</code>

¹ Here $\text{Rel}(\alpha, \beta)$ is a shorthand for $\text{Rel}(\alpha_0, \dots, \alpha_p, \beta_0, \dots, \beta_q)$ when $\alpha = \alpha_0, \dots, \alpha_p$ and $\beta = \beta_0, \dots, \beta_q$.

shares no function symbols with Σ_{Rel} . Moreover, we consider only formulas none of whose terms have a sort where Set is nested in some other constructor or itself. This is a *genuine restriction* that disallows, for instances, sorts like $\text{Set}(\text{Set}(\text{Int}))$ or $\text{List}(\text{Set}(\text{Int}))$. Extending our results so as to drop this restriction is left to future work.

B. A Calculus for $\mathbb{T}'_{\text{Rel}}$

To reason about quantifier-free formulas in $\mathbb{T}'_{\text{Rel}}$ we adopt a variant of the calculus described by Mohamed et al. [21]. Without loss of generality we assume that every set term in such formulas is *flat*, i.e., has the form $e \in s$, $s \sqcup t$, $s \sqcap t$, $s \setminus t$, $s \times t$, or $\sigma(p, s)$ where e , s and t are variables. We also assume that all equalities have the form $x \approx t$ where x is a (set or element) variable. This is possible since every quantifier-free formula φ that contains a non-flat term t can be converted into an equisatisfiable formula where every occurrence of t is replaced with a fresh variable v of the same sort, and the constraint $v \approx t$ is added as a conjunct.

To simplify the presentation, we further restrict our attention only to sets and relations over the *same* element domain, denoted generically in the following by the *element sort* ε . Our results, however, do not require this restriction, and apply more generally to constraints with sorts of the form $\text{Rel}(\tau_1, \dots, \tau_n)$.

Thanks to the following lemma, we will further focus on just sets of Σ'_{Rel} -literals, or *relation constraints*, and Σ'_{El} -literals or *element constraints*.⁴

Lemma 1. *For every quantifier-free Σ'_{Rel} -formula φ , there are sets $S_1, \dots, S_n$ of relation constraints and sets $E_1, \dots, E_n$ of element constraints such that φ is satisfiable in $\mathbb{T}'_{\text{Rel}}$ iff $S_i \cup E_i$ is satisfiable in $\mathbb{T}'_{\text{Rel}}$ for some $i \in [1, n]$.*

As a final simplification, we assume without loss of generality that for every variable x of sort $\text{Tuple}(\varepsilon, \dots, \varepsilon)$ occurring in one of the sets S_i above, S_i also contains the constraint $x \approx \langle x_0, \dots, x_k \rangle$ where $x_0, \dots, x_k$ are variables of sort ε .

⁴Detailed proofs of this lemma and later results can be found in a longer version of this paper [20].

1) *Configurations and Derivation Trees:* The calculus operates on *configurations*. These are either the distinguished configuration *unsat* or pairs (S, E) where S is a set of relation constraints and E is a set of element constraints.

A derivation rule takes a configuration and, if applicable to it, generates one or more alternative configurations. A derivation rule *applies* to a configuration C if all the conditions in the rule's premises hold for C and the rule application is not redundant. An application of a rule is *redundant* if it has a conclusion where each component in the derived configuration is a subset of the corresponding component in the premise configuration. We assume that for rules that introduce fresh variables, the introduced variables are identical whenever the premises triggering the rule are the same. In particular, we cannot generate an infinite sequence of rule applications by continuously using the current configuration to introduce fresh variables. A configuration other than *unsat* is *saturated* if every possible application of a derivation rule to it is redundant. A configuration (S, E) is *satisfiable* in $\mathbb{T}'_{\text{Rel}}$ if the set $S \cup E$ is satisfiable in $\mathbb{T}'_{\text{Rel}}$.

A *derivation tree* is a finite tree where each node is a configuration whose children, if any, are obtained by a non-redundant application of a derivation rule to the node. A derivation tree *derives from* a derivation tree T if it is obtainable from T by applying a derivation rule to one of T 's leaves. It is *closed* if all of its leaves are *unsat*.

As we show later, a closed derivation tree with root (S, E) is a proof that $S \cup E$ is unsatisfiable in $\mathbb{T}'_{\text{Rel}}$. In contrast, a derivation tree with root (S, E) and a saturated leaf is a witness that $S \cup E$ is satisfiable in $\mathbb{T}'_{\text{Rel}}$.

2) *The Derivation Rules:* The rules of our calculus are listed in Figure 1. They are expressed in *guarded assignment form* where the premise describes the conditions on the current configuration under which the rule applies, and the conclusion is either *unsat*, or otherwise describes *only the changes* to the current configuration. Rules with two conclusions, separated by the symbol $||$, are non-deterministic branching rules, in the style of analytic tableaux.

In the rules, we write S, c , as an abbreviation of $S \cup \{c\}$ and denote by $\mathcal{T}(S)$ the set of all terms and subterms occurring in S . Because of our focus on the theory $\mathbb{T}_{\text{Rel}}$ in this paper, the calculus relies on an *element oracle* that can decide the satisfiability in $\mathbb{T}'_{\text{Rel}}$ of sets of element constraints. We also require the computability of all predicates used as arguments in applications of filter (σ).

We define the following closures for S and E where $\models_{\text{tup}}$ denotes entailment in the theory of tuples, which treats all function symbols other than $\langle _ \rangle$ as uninterpreted.

$$\begin{aligned}
S^* &= \{(\neg)s \approx t \mid s, t \in \mathcal{T}(S); S \models_{\text{tup}} (\neg)s \approx t\} \\
&\quad \cup \{(\neg)e \in s \mid e, s \in \mathcal{T}(S); S \models_{\text{tup}} e \approx \hat{e} \wedge s \approx \hat{s}; (\neg)\hat{e} \in \hat{s} \in S\} \\
E^* &= \{(\neg)s \approx t \mid s, t \in \mathcal{T}(E); E \models_{\text{tup}} (\neg)s \approx t\} \\
&\quad \cup \{(\neg)p(e) \mid (\neg)p(\hat{e}) \in E; e, \hat{e} \in \mathcal{T}(E); E \models_{\text{tup}} e \approx \hat{e}\}
\end{aligned}$$

In each of these definitions, the hatted variables are implicitly existentially quantified. Note that $S^* \supseteq S$ and $E^* \supseteq E$.

$$\begin{array}{c}
\frac{t \not\approx t \in S^*}{\text{unsat}} \text{EQ UNSAT} \quad \frac{x \in s \in S^* \quad x \notin s \in S^*}{\text{unsat}} \text{SET UNSAT} \quad \frac{x \in [] \in S^*}{\text{unsat}} \text{EMPTY UNSAT} \\
\\
\frac{e_1, e_2 \in \mathcal{T}(S^*) \quad e_1, e_2 \text{ are different terms of the same element sort}}{S := S, e_1 \approx e_2 \quad E := E, e_1 \approx e_2 \quad || \quad S := S, e_1 \not\approx e_2 \quad E := E, e_1 \not\approx e_2} \text{E-IDENT} \\
\\
\frac{x \in s \in S^* \quad x \in t \in S^* \quad s \sqcap t \in \mathcal{T}(S)}{S := S, x \in s \sqcap t} \text{INTER UP} \quad \frac{x \in s \sqcap t \in S^*}{S := S, x \in s, x \in t} \text{INTER DOWN} \\
\\
\frac{x \in u \in S^* \quad u \in \{s, t\} \quad s \sqcup t \in \mathcal{T}(S)}{S := S, x \in s \sqcup t} \text{UNION UP} \quad \frac{x \in s \sqcup t \in S^*}{S := S, x \in s \quad || \quad S := S, x \in t} \text{UNION DOWN} \\
\\
\frac{x \in s \in S^* \quad s \setminus t \in \mathcal{T}(S)}{S := S, x \in t \quad || \quad S := S, x \in s \setminus t} \text{DIFF UP} \quad \frac{x \in s \setminus t \in S^*}{S := S, x \in s, x \notin t} \text{DIFF DOWN} \\
\\
\frac{[x] \in \mathcal{T}(S)}{S := S, x \in [x]} \text{SINGLE UP} \quad \frac{x \in [y] \in S^*}{S := S, x \approx y} \text{SINGLE DOWN} \\
\\
\frac{s \not\approx t \in S^*}{S := S, z \in s, z \notin t \quad || \quad S := S, z \notin s, z \in t} \text{SET DISEQ} \quad \frac{E \text{ is unsatisfiable in } \mathbb{T}'_{\text{Rel}}}{\text{unsat}} \text{E-CONF} \\
\\
\frac{\langle x_1, \dots, x_m \rangle \in r_1 \in S^* \quad \langle y_1, \dots, y_n \rangle \in r_2 \in S^* \quad r_1 \times r_2 \in \mathcal{T}(S)}{S := S, \langle x_1, \dots, x_m, y_1, \dots, y_n \rangle \in r_1 \times r_2} \text{PROD UP} \\
\\
\frac{\langle x_1, \dots, x_m, y_1, \dots, y_n \rangle \in r_1 \times r_2 \in S^* \quad \text{ar}(r_1) = m}{S := S, \langle x_1, \dots, x_m \rangle \in r_1, \langle y_1, \dots, y_n \rangle \in r_2} \text{PROD DOWN} \\
\\
\frac{e \in s \in S^* \quad \sigma(p, s) \in \mathcal{T}(S)}{E := E, p(e) \quad S := S, e \in \sigma(p, s) \quad || \quad E := E, \neg p(e) \quad S := S, e \notin \sigma(p, s)} \text{FILTER UP} \\
\\
\frac{e \in \sigma(p, s) \in S^*}{E := E, p(e) \quad S := S, e \in s} \text{FILTER DOWN}
\end{array}$$

Fig. 1. The derivation rules. Variable z in SET DISEQ is fresh. In PROD DOWN, $\text{ar}(r_1)$ denotes the arity of relation r_1 .

The sets S^* and E^* are computable from S and E by an extension of standard congruence closure procedures with rules for deducing the equalities $s_1 \approx t_1, \dots, s_n \approx t_n$ from tuple equalities of the form $\langle s_1, \dots, s_n \rangle \approx \langle t_1, \dots, t_n \rangle$.

Moving to the derivation rules, rule E-CONF is applied when a conflict is found by the element solver. The other three rules deriving *unsat* should be self-explanatory. Rule SET DISEQ handles disequality between two sets s, t by stating that some element, represented by a fresh variable z , only occurs in s or in t , but not both. Rules UNION UP, UNION DOWN, INTER UP, and INTER DOWN, DIFF UP, DIFF DOWN, SINGLE UP and SINGLE DOWN correspond directly to the semantics of their operators. PROD UP and PROD DOWN are upward and downward rules for the $\times$ operator. FILTER UP splits on whether an element e in s satisfies (the predicate denoted by) p or not in order to determine its membership in set $\sigma(p, s)$. FILTER DOWN concludes that every element in $\sigma(p, s)$ necessarily satisfies p and occurs in s .

Example 1. Suppose $\varphi = x \notin \sigma(\lambda z. z \geq x, s)$ where s has sort $\text{Set}(\text{Int})$. The initial configuration is $C_0 = (S_0, E_0)$ where $S_0 = \{x \notin \sigma(\lambda z. z \geq x, s)\}$ and $E_0 = \emptyset$. None of the rules

in Figure 1 apply to C_0 , so it is saturated and hence φ is satisfiable. As we shall see in Lemma 4, the model construction procedure will interpret x to be any integer, say 0, and s to be the empty set.

Example 2. Suppose $\varphi = x \in \sigma(\lambda z. z \geq x, s \sqcap t) \wedge x \notin s$. The initial configuration is $C_0 = (S_0, E_0)$ where $S_0 = \{x \in \sigma(\lambda z. z \geq x, s \sqcap t), x \notin s\}$ and $E_0 = \emptyset$. The premise of rule FILTER DOWN holds, and so it applies to C_0 yielding a configuration $C_1 = (S_1, E_1)$ where $S_1 = S_0 \cup \{x \in s \sqcap t\}$ and $E_1 = \{x \geq x\}$. Now rule INTER DOWN applies to C_1 yielding a configuration $C_2 = (S_2, E_2)$ where $S_2 = S_1 \cup \{x \in s, x \in t\}$ and $E_2 = E_1$. Since S_2 contains both $x \in s$ and $x \notin s$, rule SET UNSAT applies to C_2 yielding a configuration $C_3 = \text{unsat}$. From Proposition 2, the formula φ is unsatisfiable in $\mathbb{T}'_{\text{Rel}}$.

III. DECIDABILITY OF RESTRICTED FILTER

We prove that, under the conditions below, the calculus we have introduced is sound, complete and terminating.

Condition 1. No predicate in applications of the filter operator σ includes *set terms*, terms with sorts of the form $\text{Set}(\tau)$.

Condition 2. *The satisfiability of sets of element constraints is decidable.*

Condition 3. *The satisfiability of sets E of element constraints remains decidable with the addition of predicate applications generated by rules FILTER UP and FILTER DOWN.*

Note that Condition 3 follows from 2 when all filter predicates are expressed as λ -terms in the language of E .

Definition 1 (Derivation). *Let $S = S_0 \cup E_0$ be a set of $\mathbb{T}'_{\text{Rel}}$ -constraints. A **derivation** of S is a sequence $(T_i)_{0 \leq i < \kappa}$ of derivation trees, with κ finite or countably infinite, such that T_{i+1} derives from T_i for all i and T_0 is a singleton tree with root (S_0, E_0) . A refutation of S is a finite derivation of S that ends with a closed tree.*

A calculus is **terminating** if every derivation is finite.

Let $\mathcal{F}$ be the sublanguage of constraints in $\mathbb{T}'_{\text{Rel}}$ that satisfy Condition 1 and contain set operators exclusively from $\{\approx, [], [-], \sqsubseteq, \sqcup, \sqcap, \setminus, \times, \sigma\}$. Note that the omission of $\sqsubseteq$ is no real restriction since $s \sqsubseteq t$ can be expressed as $s \approx s \sqcap t$. We refer to constraints from $\mathcal{F}$ as **$\mathcal{F}$ -constraints** and restrict our attention to them in this section.

A. Termination

To start, we argue that the applicability of every rule in Figure 1 is decidable. This is trivially the case for rule E-CONF thanks to our Conditions 2 and 3. For the other rules, the argument is fairly straightforward. For example, S^* can be computed from S by an algorithm based on congruence closure, so checking the presence of constraints in S^* is decidable. Therefore, we focus on proving that the calculus has no infinite derivations when starting with a finite set of $\mathcal{F}$ -constraints.

Lemma 2. *Under Conditions 1–3, every derivation of a set of $\mathcal{F}$ -constraints is finite.*

Proof sketch. Suppose we start with a configuration $C = (S, E)$ in $\mathcal{F}$. For uniformity, and without loss of generality, suppose all set terms denote relations, i.e., sets whose elements are tuples. Now let $e_0 = |\mathcal{T}(E)|$ and $s_0 = |\mathcal{T}(S)|$ where the sets $\mathcal{T}(E)$ and $\mathcal{T}(S)$ consist of all the finitely-many terms of sort Tup and Set in S and E , respectively. Since none of the derivation rules introduces new Set terms, s_0 is constant in all derivation trees with root C . None of the rules except for SET DISEQ, PROD UP and PROD DOWN generate new element terms. (Rules FILTER UP and FILTER DOWN do not introduce new element terms because of our assumption that the predicates do not include set terms.) Since SET DISEQ can be applied only once for each disequality constraint, e_0 can increase by at most s_0^2 in all derivation trees. Let $e_1 = e_0 + s_0^2$. Rules PROD UP and PROD DOWN introduce up to e_1^2 and $2e_1$ new element terms, respectively. This means that the number of element terms in all derivation is at most $e_2 = e_1^2 + 2e_1 + e_1 = e_1^2 + 3e_1$.

We now show that every derivation starting from C is finite. We do that by defining a well-founded order $\succ$ on the set of configurations and prove that applying each rule to a

configuration $X = (S, E)$ yields a configuration Y such that $X \succ Y$. The order is defined as follows: $X \succ Y$ iff

- 1) $X \neq \text{unsat}$ and $Y = \text{unsat}$; or
- 2) $X \neq \text{unsat}$, $Y \neq \text{unsat}$ and $\langle f_1(X), \dots, f_{14}(X) \rangle \succ_p \langle f_1(Y), \dots, f_{14}(Y) \rangle$ where f_i are ranking functions defined in Table II and $\succ_p$ is the pointwise ordering extension of $>$ over the integers.

Since the conflict rules E-CONF, SET UNSAT, EMPTY UNSAT, and EQ UNSAT derive the configuration unsat , which is a minimal element of $\succ$, the claim $X \succ Y$ is immediate for them. Each application of the other rules only reduces the value of its corresponding ranking function in Table II and does not increase the value of the other functions. For these rules too we then have that $X \succ Y$. To conclude the proof we only have to argue that $\succ$ is well-founded but this is a consequence of the fact that, in any given derivation tree, every function in Table II is bounded below by 0. $\square$

Termination is a direct consequence of Lemma 2.

Proposition 1. *Under Conditions 1–3, the calculus is terminating over sets of $\mathcal{F}$ -constraints.*

B. Refutation Soundness

The refutation soundness of the calculus follows from the fact that all of its derivation rules preserve equisatisfiability.

Lemma 3. *For every rule of the calculus, the premise configuration is equisatisfiable in $\mathbb{T}'_{\text{Rel}}$ with the disjunction of its conclusions.*

Proof. The proof follows from the semantics of set operators and the definition of S^* and E^* . We show the proof for rules FILTER DOWN and FILTER UP. The other cases are similar. Suppose the premise configuration is $C_1 = (S_1, E_1)$. Rule FILTER DOWN has only one conclusion, say $C_2 = (S_1 \cup \{e \in s\}, E_1 \cup \{p(e)\})$. We prove a stronger claim than equisatisfiability that any model satisfies C_1 iff it satisfies C_2 . Suppose $\mathcal{I}_1$ is a model for C_1 . We show that $\mathcal{I}_1$ is also a model for C_2 that satisfies the new constraints $e \in s$ and $p(e)$. The rule is applied because $e' \in \sigma(p, s') \in S_1$ for some $e', s' \in \mathcal{T}(S_1)$ such that $e \approx e'$ and $s \approx s'$ in S_1^* . So, $\mathcal{I}_1$ satisfies $e \in \sigma(p, s)$ and $\mathcal{I}_1(e) \in \mathcal{I}_1(\sigma(p, s))$. From the semantics of $\sigma(p, s)$, it must be the case that $\mathcal{I}(p(e))$ holds and $\mathcal{I}(e) \in \mathcal{I}(s)$. Therefore $\mathcal{I}_1$ satisfies $e \in s$ and $p(e)$, and hence satisfies C_2 . For the other direction, suppose $\mathcal{I}_2$ satisfies C_2 . Since $S_1 \subseteq S_1 \cup \{e \in s\}$ and $E_1 \subseteq E_1 \cup \{p(e)\}$, then $\mathcal{I}_2$ satisfies $C_1 = (S_1, E_1)$. So, rule FILTER DOWN is sound.

For rule FILTER UP, suppose the premise configuration $C_1 = (S_1, E_1)$ is satisfied by a model $\mathcal{I}_1$. In that model, we have either $\mathcal{I}(p(e))$ holds or $\neg \mathcal{I}(p(e))$ holds. If $\mathcal{I}(p(e))$ holds, then $\mathcal{I}_1$ is a model for the first branch of the conclusion, because $\mathcal{I}(e) \in \mathcal{I}(\sigma(p, s))$. If $\neg \mathcal{I}(p(e))$ holds, then $\mathcal{I}_1$ is a model for the second branch, since $\mathcal{I}(e) \notin \mathcal{I}(\sigma(p, s))$. Now suppose $\mathcal{I}_2$ is a model for one of the two possible configurations $C_2 = (S_1 \cup \{e \in \sigma(p, s)\}, E_1 \cup \{p(e)\})$ or $C'_2 = (S_1 \cup \{e \notin \sigma(p, s)\}, E_1 \cup \{\neg p(e)\})$ in the conclusion. Since $e \in s \in S_1^*$ in C_2 and C'_2 , then $\mathcal{I}_2$ satisfies $e \in s$. So, $\mathcal{I}_2$ is a model for C_1 . $\square$

TABLE II
RANKING FUNCTIONS FOR RELATION RULES. HERE, $\bar{x} = \langle x_1, \dots, x_m \rangle$, $\bar{y} = \langle y_1, \dots, y_n \rangle$, AND $\langle \bar{x}, \bar{y} \rangle$ DENOTES $\langle x_1, \dots, x_m, y_1, \dots, y_n \rangle$.

f_i	Rule	Definition
f_1, f_2	INTER UP, INTER DOWN	$e_2 \cdot s_0^2 - \{x \in s \cap t \mid x \in s \cap t \in S\} $, $e_2 \cdot s_0^2 - \{x \in s \mid x \in s \in S\} $
f_3, f_4	UNION UP, UNION DOWN	$e_2 \cdot s_0^2 - \{x \in s \sqcup t \mid x \in s \sqcup t \in S\} $, $e_2 \cdot s_0^2 - \{x \in s \mid x \in s \in S\} $
f_5, f_6	DIFF UP, DIFF DOWN	$e_2 \cdot s_0^2 - \{x \in s \mid x \in s \in S\} $, $e_2 \cdot s_0^2 - \{x \notin s \mid x \notin s \in S\} $
f_7, f_8	SINGLE UP, SINGLE DOWN	$s_0 - \{x \in [x] \mid x \in [x] \in S\} $, $e_2 \cdot s_0 - \{x \approx y \mid x \approx y, x \in [y] \in S\} $
f_9	SET DISEQ	$s_0^2 - \{(z_{s,t} \in s, z_{s,t} \notin t) \mid z_{s,t} \in s, z_{s,t} \notin t, s \not\approx t \in S\} $
f_{10}, f_{11}	PROD UP, PROD DOWN	$e_2^2 \cdot s_0^2 - \{(\bar{x}, \bar{y}) \mid \langle \bar{x}, \bar{y} \rangle \in s \times t \in S\} $, $e_2^2 \cdot s_0^2 - \{(\bar{x}, \bar{y}) \mid \bar{x} \in s \in S, \bar{y} \in t \in S\} $
f_{12}	E-IDENT	$e_2^2 - \{e_1 \approx e_2 \mid e_1 \approx e_2 \in S\} - \{e_1 \not\approx e_2 \mid e_1 \approx e_2 \in S\} $
f_{13}	FILTER UP	$e_2 \cdot s_0 - \{(p(e), e \in \sigma(p, s)) \mid p(e) \in E, e \in \sigma(p, s) \in S\} $ $- \{(\neg p(e), e \notin \sigma(p, s)) \mid \neg p(e) \in E, e \notin \sigma(p, s) \in S\} $
f_{14}	FILTER DOWN	$e_2 \cdot s_0 - \{(p(e), e \in s) \mid p(e) \in E, e \in s \in S\} $

Proposition 2 (Soundness). *Every set of $\mathcal{F}$ -constraints that has a refutation is unsatisfiable in $\mathbb{T}'_{\text{Rel}}$.*

Proof. Given that, by Lemma 3, every rule preserves constraint equisatisfiability, we can argue by structural induction on derivation trees that the root of any closed derivation tree is unsatisfiable in $\mathbb{T}'_{\text{Rel}}$. The claim then holds because every refutation of a set S of $\mathcal{F}$ -constraints starts with a configuration that is equisatisfiable with S in $\mathbb{T}'_{\text{Rel}}$. $\square$

C. Refutation Completeness

Proposition 3 (Completeness). *Under Conditions 1–3, if $S = S_0 \cup E_0$ is a set of $\mathcal{F}$ -constraints unsatisfiable in $\mathbb{T}'_{\text{Rel}}$, then it has a refutation.*

Proof. We prove the contrapositive. Suppose S does not have a refutation. Since the calculus is terminating (Proposition 1), it has only finite derivations. Let D be the longest possible derivation starting from S . Then, since D is not a refutation and no extension is possible, it must end with a tree with a saturated leaf (S_1, E_1) . By Lemma 4 (below), $S_1 \cup E_1$ is then satisfiable in $\mathbb{T}'_{\text{Rel}}$. The satisfiability of S in $\mathbb{T}'_{\text{Rel}}$ follows from that fact that $S \subseteq S_1 \cup E_1$, which can be proven by structural induction on derivation trees. $\square$

The main step in the proof of the proposition above relies on the following result where $\text{Vars}(A)$ denotes all the variables occurring in some expression in the set A .

Lemma 4. *Let $S_0 \cup E_0$ be a set of $\mathcal{F}$ -constraints where S_0 and E_0 consist of set and element constraints, respectively. Suppose D is a finite derivation of (S_0, E_0) . If its final tree has a saturated leaf $C_1 = (S_1, E_1)$, then there exists a model $\mathcal{I}$ of $\mathbb{T}'_{\text{Rel}}$ that satisfies $S_1 \cup E_1$ and has the following properties:*

- 1) For all $x, y \in \text{Vars}(S_1) \cup \text{Vars}(E_1)$ of element sort, $\mathcal{I}(x) = \mathcal{I}(y)$ iff $x \approx y \in E_1^*$.
- 2) For all $s \in \text{Vars}(S_1)$ of set sort, $\mathcal{I}(s) = \{\mathcal{I}(x) \mid x \in s \in S_1^*\}$.

Proof. For simplicity, we consider only initial configurations (S_0, E_0) that contain no variables of sorts of the form $\text{Typ}(\alpha_1, \dots, \alpha_n)$ since any such variable can be replaced by a

tuple $\langle x_1, \dots, x_n \rangle$ where each x_i is a variable of sort α_i . We assume, with no loss of generality, that all set equalities in S_1 have the form $x \approx t$ where x is a variable that never occurs in the right-hand side of an equality of S_1 . This restriction allows us to define a well-founded ordering $\succ_T$ over the set variables in S_1 where $x \succ_T y$ iff $x \approx t \in S_1$ and y occurs in t .

Since the leaf (S_1, E_1) is saturated, rules E-IDENT, FILTER DOWN and FILTER UP do not apply. This implies that all predicates and equalities between elements generated by the filter rules have been propagated to component E_1 . Since rule E-CONF does not apply either, we can conclude that E_1 is satisfiable in $\mathbb{T}'_{\text{Rel}}$. Let $\mathcal{I}$ be any interpretation that satisfies E_1 . It is possible to show that $\mathcal{I}$ satisfies Property (1).

We modify $\mathcal{I}$ so that it assigns to each set variable s in S_1 the value $\{\mathcal{I}(e) \mid e \in s \in S_1^*\}$. Since E_1 contains no set variables, $\mathcal{I}$ continues to satisfy E_1 . We argue by induction on $\succ_T$ that $\mathcal{I}$ satisfies every constraint c in S_1 as well.

If c is a membership constraint, we know c has the form $e \in s$ where e and s are variables. Then we have that $\mathcal{I}(e) \in \mathcal{I}(s)$ by construction of $\mathcal{I}$ because $S_1 \subseteq S_1^*$. If c is an equality, we know c has either the form $e_1 \approx e_2$ where e_1 is an element variable or the form $s \approx t$ where s is a set variable. In the first case, thanks to rule E-IDENT and saturation $e_1 \approx e_2$ is in E_1 and so $e_1 \approx e_2$ is satisfied by $\mathcal{I}$ by Property (1). In the second case, we have the following subcases for t :

1) $s \approx t$ where t is a variable. We have $\mathcal{I}(s) = \mathcal{I}(t)$ because then, for each element variable e , $e \in s \in S_1^*$ iff $e \in t \in S_1^*$ by the definition of S_1^* .

2) $s \approx \sigma(p, t)$ where s and t are set variables. Observe that $s \succ_T t$ and let $A = \{a \mid a \in \mathcal{I}(t), \mathcal{I}(p)(a) = \text{true}\}$. By the semantics of σ , we need to show that $\mathcal{I}(s) = A$. Suppose $a \in \mathcal{I}(s)$. By the definition of $\mathcal{I}$, we have $a = \mathcal{I}(e)$ for some e with $e \in s \in S_1^*$. By the definition of S_1^* , we have $e \in \sigma(p, t) \in S_1^*$. Now $e \in t \in S_1, p(e) \in E_1$ because of saturation wrt rule FILTER DOWN. By induction on t and the fact that $\mathcal{I}$ satisfies E_1 , we have $a \in \mathcal{I}(t)$ and $\mathcal{I}(p)(a)$ is true. Therefore $a \in A$. For the other direction, suppose $a \in A$. That means $a \in \mathcal{I}(t)$ and $\mathcal{I}(p)(a)$ is true. From the induction hypothesis on t , we have $a = \mathcal{I}(e)$ for some $e \in$

$t \in S_1^*$. By saturation wrt rule FILTER UP, E_1 must contain one of the rule's conclusions. The second one cannot be, since $\neg p(e) \in E_1$ implies $\neg \mathcal{I}(p(e))$ which means $\mathcal{I}(p)(a)$ is false, a contradiction. So E_1 must contain the first conclusion of FILTER UP. But then we have $p(e) \in E_1, e \in \sigma(p, t) \in S_1$. This means that $a = \mathcal{I}(e), e \in s \in S_1^*$. From the definition of $\mathcal{I}(s)$, we have $a \in \mathcal{I}(s)$.

3) $s \approx \square$. Rule EMPTY UNSAT would apply to the configuration if there was a constraint of the form $x \in s \in S_1^*$. Since there is none, it follows that $\mathcal{I}(s) = \{\} = \mathcal{I}(\square)$.

4) $s \approx [x]$. It is sufficient to show that $\mathcal{I}(s) = \{\mathcal{I}(x)\}$. Since rule SINGLE UP is not applicable, we can conclude that $x \in s \in S_1^*$. It follows that $\{\mathcal{I}(x)\} \subseteq \mathcal{I}(s)$. The other direction, $\mathcal{I}(s) \subseteq \{\mathcal{I}(x)\}$, follows because of saturation with respect to rule SINGLE DOWN.

5) $s \approx t \sqcap u$. We need to show that $\mathcal{I}(s) = \mathcal{I}(t) \cap \mathcal{I}(u)$. The proof of the left-to-right inclusion depends on rule INTER DOWN. Suppose $a \in \mathcal{I}(s)$. We have:

$$\begin{aligned} a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in s \in S_1^* && \text{(definition of } \mathcal{I}(s)) \\ a &= \mathcal{I}(e), e \in t \sqcap u \in S_1^* && \text{(definition of } S_1^* \text{ and } s \approx t \sqcap u) \\ a &= \mathcal{I}(e), e \in t \in S_1, e \in u \in S_1 && \text{(rule INTER DOWN)} \\ a &\in \mathcal{I}(t) \text{ and } a \in \mathcal{I}(u) \text{ (by ind. hypothesis since } s \succ_T t, u) \\ a &\in \mathcal{I}(t) \cap \mathcal{I}(u). \end{aligned}$$

For the other direction, we rely on rule INTER UP. Suppose $a \in \mathcal{I}(t) \cap \mathcal{I}(u)$. We have:

$$\begin{aligned} a &\in \mathcal{I}(t) \text{ and } a \in \mathcal{I}(u) \\ a &= \mathcal{I}(e_1) = \mathcal{I}(e_2) \text{ for some } e_1 \in t \in S_1^*, e_2 \in u \in S_1^* \\ &\quad \text{(by induction hypothesis since } s \succ_T t \text{ and } s \succ_T u) \\ a &= \mathcal{I}(e_1) = \mathcal{I}(e_2), e_1 \in t \in S_1^*, e_2 \in u \in S_1^*, e_1 \approx e_2 \in S_1^* \\ &\quad \text{(saturation of rule E-IDENT)} \\ a &= \mathcal{I}(e_1), e_1 \in t \in S_1^*, e_1 \in u \in S_1^* && \text{(definition of } S_1^*) \\ a &= \mathcal{I}(e_1), e_1 \in t \sqcap u \in S_1^* && \text{(rule INTER UP)} \\ a &= \mathcal{I}(e_1), e_1 \in s \in S_1^* && (s \approx t \sqcap u) \\ a &\in \mathcal{I}(s). && \text{(definition of } \mathcal{I}(s)) \end{aligned}$$

Note that if we were to choose the second conclusion in rule E-IDENT, $e_1 \not\approx e_2 \in E_1$ would contradict that $\mathcal{I}(e_1) = \mathcal{I}(e_2)$.

6) $s \approx t \sqcup u$. We need to show that $\mathcal{I}(s) = \mathcal{I}(t) \cup \mathcal{I}(u)$. The proof of the left-to-right inclusion depends on rule INTER DOWN. Suppose $a \in \mathcal{I}(s)$. We have:

$$\begin{aligned} a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in s \in S_1^* && \text{(definition of } \mathcal{I}(s)) \\ a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in t \sqcup u \in S_1^* \\ &\quad \text{(definition of } S_1^* \text{ and } s \approx t \sqcup u) \\ a &= \mathcal{I}(e), e \in t \in S_1 \text{ or } e \in u \in S_1 && \text{(rule UNION DOWN)} \\ a &\in \mathcal{I}(t) \text{ or } a \in \mathcal{I}(u) \text{ (by ind. hypothesis since } s \succ_T t, u) \\ a &\in \mathcal{I}(t) \cup \mathcal{I}(u). \end{aligned}$$

For the other direction, suppose $a \in \mathcal{I}(t) \cup \mathcal{I}(u)$. We have:

$$\begin{aligned} a &\in \mathcal{I}(t) \text{ or } a \in \mathcal{I}(u) \\ a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in t \in S_1^* \text{ or } e \in u \in S_1^* \\ &\quad \text{(by induction hypothesis since } s \succ_T t \text{ and } s \succ_T u) \\ a &= \mathcal{I}(e), e \in t \sqcup u \in S_1^* && (t \sqcup u \in \mathcal{T}(S_1), \text{ rule UNION UP}) \\ a &= \mathcal{I}(e), e \in s \in S_1^* && (s \approx t \sqcup u) \\ a &\in \mathcal{I}(s). && \text{(definition of } \mathcal{I}(s)) \end{aligned}$$

7) $s \approx t \setminus u$. We need to show that $\mathcal{I}(s) = \mathcal{I}(t) \setminus \mathcal{I}(u)$. The proof of the left-to-right inclusion depends on rule DIFF DOWN. Suppose $a \in \mathcal{I}(s)$. We have:

$$\begin{aligned} a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in s \in S_1^* && \text{(definition of } \mathcal{I}(s)) \\ a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in t \setminus u \in S_1^* \\ &\quad \text{(definition of } S_1^* \text{ and } s \approx t \setminus u) \\ a &= \mathcal{I}(e), e \in t \in S_1^* \text{ and } e \notin u \in S_1^* && \text{(rule DIFF DOWN)} \\ a &\in \mathcal{I}(t) \text{ and } a \notin \mathcal{I}(u) \\ &\quad \text{(by induction (} s \succ_T t \text{) and contradiction below)} \\ a &\in \mathcal{I}(t) \setminus \mathcal{I}(u). \end{aligned}$$

Suppose by contradiction that $a \in \mathcal{I}(u)$. Since $s \succ_T u$, by induction hypothesis there exists e_2 such that $a = \mathcal{I}(e) = \mathcal{I}(e_2)$ and $e_2 \in u \in S_1^*$. By saturation of E-IDENT either $e \approx e_2 \in S_1^* \cap E_1^*$, or $e \not\approx e_2 \in S_1^* \cap E_1^*$. The latter would lead to a contradiction because $\mathcal{I}(e) = \mathcal{I}(e_2)$. The former would imply $e \in u \in S_1^*$ which would trigger rule SET UNSAT along with $e \notin u \in S_1^*$ to make the current branch unsat. This contradicts our assumption that the current branch is open.

For the other direction, suppose $a \in \mathcal{I}(t) \setminus \mathcal{I}(u)$. We have:

$$\begin{aligned} a &\in \mathcal{I}(t) \text{ and } a \notin \mathcal{I}(u) \\ a &= \mathcal{I}(e) \text{ for some } e \text{ with } e \in t \in S_1^* \\ &\quad \text{(by induction hypothesis since } s \succ_T t) \\ a &= \mathcal{I}(e), e \in t \setminus u \in S_1^* \\ &\quad (t \setminus u \in \mathcal{T}(S_1), \text{ rule DIFF UP is discussed below)} \\ a &= \mathcal{I}(e), e \in s \in S_1^* && (s \approx t \setminus u) \\ a &\in \mathcal{I}(s) && \text{(definition of } \mathcal{I}(s)) \end{aligned}$$

Rule DIFF UP splits into two branches. One with $e \in u \in S_1^*$ and the other is the one we mentioned above $e \in t \setminus u \in S_1^*$. Suppose by contradiction that the current branch has $e \in u \in S_1^*$. Since $s \succ_T u$, by our induction hypothesis we have $a \in \mathcal{I}(u)$ which contradicts $a \notin \mathcal{I}(u)$.

8) $s \approx t \times u$. We need to show that $\mathcal{I}(s) = \mathcal{I}(t) \times \mathcal{I}(u)$. We use the notation: $\bar{x}$ for $\langle x_1, \dots, x_m \rangle$, $\bar{y}$ for $\langle y_1, \dots, y_n \rangle$, $\langle \bar{x}, \bar{y} \rangle$ for $\langle x_1, \dots, x_m, y_1, \dots, y_n \rangle$, $\mathcal{I}(\bar{x})$ for $\langle \mathcal{I}(x_1), \dots, \mathcal{I}(x_m) \rangle = \langle a_1, \dots, a_m \rangle$, $\mathcal{I}(\bar{y})$ for $\langle \mathcal{I}(y_1), \dots, \mathcal{I}(y_n) \rangle = \langle b_1, \dots, b_n \rangle$, and $\mathcal{I}(\langle \bar{x}, \bar{y} \rangle)$ for $\langle \mathcal{I}(x_1), \dots, \mathcal{I}(x_m), \mathcal{I}(y_1), \dots, \mathcal{I}(y_n) \rangle$. The

proof of the left-to-right inclusion depends on rule PROD DOWN. Suppose $\langle a_1, \dots, a_m, b_1, \dots, b_n \rangle \in \mathcal{I}(s)$. We have:

$$\langle a_1, \dots, a_m, b_1, \dots, b_n \rangle = \mathcal{I}(\langle \bar{x}, \bar{y} \rangle) \text{ for some } \langle \bar{x}, \bar{y} \rangle \in s \in S_1^* \quad (\text{definition of } \mathcal{I}(s))$$

$$\langle \bar{x}, \bar{y} \rangle \in t \times u \in S_1^* \quad (\text{definition of } S_1^* \text{ and } s \approx t \times u)$$

$$\bar{x} \in t \in S_1, \bar{y} \in u \in S_1 \quad (\text{rule PROD DOWN})$$

$$\langle a_1, \dots, a_m \rangle \in \mathcal{I}(t), \langle b_1, \dots, b_n \rangle \in \mathcal{I}(u) \quad (\text{by induction hypothesis since } s \succ_T t \text{ and } s \succ_T u)$$

$$\langle a_1, \dots, a_m, b_1, \dots, b_n \rangle \in \mathcal{I}(t) \times \mathcal{I}(u).$$

For the other direction, suppose $\langle a_1, \dots, a_m, b_1, \dots, b_n \rangle \in \mathcal{I}(t) \times \mathcal{I}(u)$. We have:

$$\langle a_1, \dots, a_m \rangle \in \mathcal{I}(t), \langle b_1, \dots, b_n \rangle \in \mathcal{I}(u)$$

$$\langle a_1, \dots, a_m \rangle = \mathcal{I}(\bar{x}), \langle b_1, \dots, b_n \rangle = \mathcal{I}(\bar{y})$$

$$\text{for some } \langle x_1, \dots, x_m \rangle \in t, \langle y_1, \dots, y_n \rangle \in u \in S_1^* \quad (\text{by induction hypothesis since } s \succ_T t \text{ and } s \succ_T u)$$

$$\langle x_1, \dots, x_m, y_1, \dots, y_n \rangle \in t \times u \in S_1^* \quad (t \times u \in \mathcal{T}(S_1), \text{ rule PROD UP})$$

$$\langle x_1, \dots, x_m, y_1, \dots, y_n \rangle \in s \in S_1^* \quad (s \approx t \times u)$$

$$\langle \mathcal{I}(x_1), \dots, \mathcal{I}(x_m), \mathcal{I}(y_1), \dots, \mathcal{I}(y_n) \rangle \in \mathcal{I}(s) \quad (\text{definition of } \mathcal{I}(s))$$

$$\langle a_1, \dots, a_m, b_1, \dots, b_n \rangle \in \mathcal{I}(s). \quad (\text{Property 1})$$

This concludes the proof for equality constraints. We now argue that $\mathcal{I}$ satisfies negative membership and disequality constraints. Let $e \notin s \in S_1$. By contradiction, suppose that $\mathcal{I}$ does not satisfy this constraint, i.e., $\mathcal{I}(e) \in \mathcal{I}(s)$. Then, by construction of $\mathcal{I}$, it must be the case that $e \in s \in S_1^*$. But then rule SET UNSAT applies, which contradicts our assumption that (S_1, E_1) is saturated.

For constraints of the form $s \not\approx t \in S_1^*$, s and t could be either element terms or set terms. In the first case, suppose that $\mathcal{I}(s) = \mathcal{I}(t)$, then it must be that $s \approx t \in S_1^*$ which makes EQ UNSAT applicable, again against the assumption that our leaf is saturated. If the second case, since rule SET DISEQ does not apply because the leaf is saturated, it must be that one of its conclusions, $z \in s, z \notin t$ or $z \in t, z \notin s$, is in S_1 . But then $\mathcal{I}(z)$ cannot be in both $\mathcal{I}(s)$ and $\mathcal{I}(t)$, which means that $\mathcal{I}(s) \neq \mathcal{I}(t)$. $\square$

Corollary 1. *The satisfiability in $\mathbb{T}'_{\text{Rel}}$ of $\mathcal{F}$ -constraints is decidable.*

D. Undecidability with Unrestricted Filter Predicates

One may wonder if the restriction expressed by Condition 1 is really necessary for the decidability result in the previous section. The answer is that some restriction on the predicates passed to filter is indeed needed because otherwise decidability is lost. We show that in this section at a high level by sketching a reduction from a known undecidable problem. To start, we show that having unrestricted filter predicates allows us to

define a set map operator π , which takes a function f of rank $\alpha \rightarrow \beta$ and a finite set t of sort $\text{Set}(\alpha)$, and returns the image of t under f . We can do that in sets of constraints by replacing every term of the form $\pi(f, t)$ by a fresh variable s and adding the two constraints:

$$t \approx [x \mid x \in t \wedge f(x) \in s]$$

$$s \approx [y \mid y \in s \wedge [x \mid x \in t \wedge f(x) \approx y] \not\approx []]$$

i.e., in desugared notation:

$$t \approx \sigma(\lambda x. f(x) \in s, t) \quad s \approx \sigma(\lambda y. \sigma(\lambda x. y \approx f(x), t) \not\approx [], s)$$

The first constraint ensures that each element in t has an image under f in s , whereas the second ensures that each element in s has a preimage under f in t . Next, we show the satisfiability of constraints with map terms is undecidable.

The necessity of Condition 2 and 3 for decidability should be clear, so here we focus on the necessity of Condition 1. We consider a particular element theory $\mathbb{T}_{\text{E}}$ that satisfies conditions 2 and 3, and demonstrate the undecidability of $\mathbb{T}'_{\text{Rel}}$ when Condition 1 is dropped. Specifically, we consider as $\mathbb{T}_{\text{E}}$ the combination of linear integer arithmetic (LIA) and the theory of pairs, i.e., tuples of two elements.

The decidability of the satisfiability of quantifier-free formulas in this combined theory can be proven by standard theory combination results [23], [28]. In fact, it is well-known that (LIA) has a decidable quantifier-free satisfiability problem and is stably infinite.⁵ The same is true for the theory of pairs, which is stably infinite over each of its element sorts. This is a consequence of more general results about the theory of algebraic datatypes [3], [27]. As a result, the combination of these two theories, the theory of integers and integer pairs with no multiplication, has a decidable quantifier-free satisfiability problem. We will use this combined theory as the element theory for our theory of sets and show that the satisfiability of set constraints containing applications of π to functions from LIA and pairs is undecidable. We do that by reducing Hilbert's tenth problem to formulas in a language we call $\mathcal{L}_\pi$ that only accepts integers and integer pairs as elements, and only allows arithmetic operators inside filter predicates. Figure 2 shows the grammar of $\mathcal{L}_\pi$.

For Hilbert's tenth problem, it is enough to only consider the system of equations described in Cantone et al. [4]: $x \approx y, x \approx y + z, x \approx y \cdot z, x \approx k$, where x, y, z are nonnegative integers and k is a constant. Let $\mathcal{H}$ denote the language of these equations. The syntax of $\mathcal{L}_\pi$ allows us to define x, y, z, k as elements (integers), and use arithmetic expressions in the first argument of map terms, but it does not allow us to use addition and multiplication between element variables. However, we can encode them with other operators in $\mathcal{L}_\pi$, as we show next.

To each integer variable v we associate a set variable s_v . Then we add the two constraints: $s_x \approx [x]$ and $s_x \approx \pi(\lambda n. \text{ite}(n \geq 0, n, -n), s_x)$, where ite is the if-then-else operator. Every interpretation $\mathcal{I}$ that satisfies the two constraints

⁵A Σ -theory T is *stably infinite* over a sort α in Σ if every quantifier-free Σ -formula satisfiable in T is satisfiable in a model of T that interprets α as an infinite set.

Set Formula	φ	$::= e \in s \mid s_1 \sqsubseteq s_2 \mid e_1 \approx e_2 \mid s_1 \approx s_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg \varphi_1$
Element term	e	$::= k \mid x \mid \langle x_1, x_2 \rangle$
Set term	s	$::= x \mid [e] \mid s_1 \sqcup s_2 \mid s_1 \sqcap s_2 \mid \pi(e^\lambda, s_1)$
Lambda term	e^λ	$::= \lambda x. e^{\text{LIA}} \mid \lambda \langle x_1, x_2 \rangle. e^{\text{LIA}} \mid \lambda \langle x_1, x_2 \rangle. \langle e_1^{\text{LIA}}, e_2^{\text{LIA}} \rangle$
LIA term	e^{LIA}	$::= k \mid x \mid k \cdot x \mid e_1^{\text{LIA}} + e_2^{\text{LIA}} \mid -e_1^{\text{LIA}} \mid \text{ite}(\varphi^{\text{LIA}}, e_1^{\text{LIA}}, e_2^{\text{LIA}})$
LIA constraint	φ^{LIA}	$::= e_1^{\text{LIA}} \approx e_2^{\text{LIA}} \mid e_1^{\text{LIA}} > e_2^{\text{LIA}} \mid e_1^{\text{LIA}} \geq e_2^{\text{LIA}} \mid \varphi_1^{\text{LIA}} \wedge \varphi_2^{\text{LIA}} \mid \varphi_1^{\text{LIA}} \vee \varphi_2^{\text{LIA}} \mid \neg \varphi_1^{\text{LIA}}$

Fig. 2. $\mathcal{L}_\pi$ grammar.

interprets s_x as the singleton set $\{\mathcal{I}(x)\}$ where $\mathcal{I}(x)$ is a nonnegative integer. For constraints in $\mathcal{H}$ of the form $x \approx y$ and $x \approx k$, we add to $\mathcal{H}$ the constraints $s_x \approx s_y$ and $s_x \approx [k]$, respectively. For each constraint in $\mathcal{H}$ of the form $x \approx y + z$, we add the constraint $s_x \approx \pi(\lambda n. y + n, s_z)$.

For each constraint $x \approx y \cdot z$ in $\mathcal{H}$ we add an integer variable $v_{y,z}$, a set variable $p_{y,z}$, and the clause

$$(y \approx 0 \wedge s_x \approx [0]) \vee (z \approx 0 \wedge s_x \approx [0]) \vee (y \not\approx 0 \wedge z \not\approx 0 \wedge \varphi)$$

where φ is the conjunction of the following constraints:

$$\begin{aligned} s_x &\approx [v_{y,z}] \\ [v_{y,z}] &\approx \pi(\lambda(m, n). \text{ite}(m \approx 1, n, v_{y,z}), p_{y,z}) \\ p_{y,z} &\approx [(y, z)] \sqcup \pi(\lambda(a, b). \text{ite}(a \approx 1, (a, b), (a - 1, b + z)), p_{y,z}) \end{aligned}$$

Now, any interpretation that satisfies the system of equations in $\mathcal{H}$ defines one that satisfies the corresponding constraints in $\mathcal{L}_\pi$ by interpreting the variables as follows:

$$\begin{aligned} \mathcal{I}(s_x) &= \{\mathcal{I}(v_{y,z})\} & \mathcal{I}(v_{y,z}) &= \mathcal{I}(y) \cdot \mathcal{I}(z) \\ \{\mathcal{I}(y), \mathcal{I}(z), (\mathcal{I}(y) - 1, 2 \cdot \mathcal{I}(z)), \dots, (1, \mathcal{I}(y) \cdot \mathcal{I}(z))\} &\subseteq \mathcal{I}(p_{y,z}) \end{aligned}$$

Note that $\mathcal{I}(x)$ must be nonnegative due to our constraints. Due to the undecidability of Hilbert's tenth problem [17], we can conclude that the satisfiability in $\mathbb{T}_{\text{Rel}}$ of formulas in the fragment $\mathcal{L}_\pi$ is undecidable.

IV. BOUNDED SET QUANTIFIERS

Applications that can encode their problems as formulas in $\mathbb{T}'_{\text{Rel}}$ typically require the use of quantifiers. However, in most cases all quantified variables are *bounded* in the sense that they are constrained to range over a specific finite set.

Now, the theory $\mathbb{T}'_{\text{Rel}}$ has the nice property of being able to express bounded quantification at the quantifier-free level. In fact, since formulas in this theory are just Boolean terms, every formula of the form $\exists x. (x \in s \wedge \varphi)$ can be expressed equivalently as the constraint $\sigma(\lambda x. \varphi, s) \not\approx []$. Similarly, every formula of the form $\forall x. (x \in s \Rightarrow \varphi)$ can be expressed equivalently as the constraint $\sigma(\lambda x. \varphi, s) \approx s$.

Since the SMT solver *cvc5* already supports an extension of $\mathbb{T}'_{\text{Rel}}$, we extended its language further with the set-bounded quantifier operators *set.some* and *set.all* defined internally as:

$$\text{set.some}(p, s) \equiv \sigma(p, s) \not\approx [] \quad \text{set.all}(p, s) \equiv \sigma(p, s) \approx s$$

for all predicates p of rank $\alpha \rightarrow \text{Bool}$ and sets s of sort $\text{Set}(\alpha)$.

Referring back to the decidable fragment $\mathcal{F}$ defined in Section III, notice that formulas of the form

$\text{set.all}(\dots(\text{set.all}(p, s_n))\dots, s_1)$ fall outside that fragment. However, since the theory $\mathbb{T}'_{\text{Rel}}$ supports the cross-product operator among sets (yielding a set of tuples) it is possible to rewrite nested set-bounded quantifiers that are all universal into constraints of the form $\text{set.all}(p, s_1 \times \dots \times s_n)$ where p ranges over tuples. Such constraints do fall in $\mathcal{F}$, which means that we can effectively express in $\mathcal{F}$, and hence decide, universal formulas with set-bounded quantifiers. This result is in line with the decidability of the (unsorted) logic MLSSF^V , which only allows set-bounded universal quantifiers [8].

Regardless of decidability considerations, the ability to express bounded quantification in an SMT solver without actually using standard quantifiers is rather enticing since the performance of SMT solvers is notoriously fickle in the presence of quantifiers in input formulas.

We investigated the potential of set-bounded quantifiers with an experimental evaluation that considers formulas generated by the SLEEC tool. We discuss this investigation next.

V. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented the calculus described earlier in the *cvc5* solver [2] and evaluated the implementation against problems derived from SLEEC models. SLEEC is an event-based formal language for writing Social, Legal, Ethical, Empathetic, and Cultural requirements [13]. It aims to formalize requirements in such domains as customer service, healthcare, and education. Users can write them as intuitive high-level rules in SLEEC, and then use automated tools to identify conflicts, redundancies, and other issues.

The tool LEGOS-SLEEC in particular translates SLEEC requirements into the logic FOL^* of quantified formulas over relational objects, where a relational object with n attributes of a relational class C represents an n -ary tuple in a set s_C [12]. The tool checks the satisfiability of FOL^* formulas using LEGOS, a custom-built bounded satisfiability checker [11]. LEGOS incrementally expands a quantified FOL^* formula within a fixed domain of relational objects, generating quantifier-free over- and under-approximations of the formula. The satisfiability of these approximations is determined using the SMT solver *z3* [22]. When the over-approximation is satisfiable but the under-approximation is unsatisfiable, LEGOS computes a *minimal correction set* for the under-approximation based on the difference between the two approximations. This set is used to incrementally expand the domain of the bounded variables, enabling a new iteration of the analysis.

TABLE III
SLEEC PERFORMANCE (20s timeout).

Technique	Solver	sat	unsat	unknown
0 LEGOS	z3	1204	158	0
1 $\forall/\exists$ pred	z3	282	158	922
2 $\forall/\exists$ pred-mbqi	cvc5	249	73	1040
3 $\forall/\exists$ sets-enum	cvc5	0	147	1215
4 $\forall/\exists$ sets-fmf	cvc5	486	1	875
5 set.all/set.some	cvc5	1189	138	35

Row 0 in Table III shows the performance of this custom algorithm over a large set of SLEEC benchmarks [19]. We consider it as the baseline for comparison with the quantified approaches described next. The authors of LEGOS aimed to simplify their FOL* satisfiability checking algorithm for simplicity and maintainability by directly using quantifiers in first-order logic (FOL). FOL* formulas can be translated into FOL by mapping relational classes to either sets or uninterpreted predicates. Specifically, a class C of relational objects with n attributes in FOL* can be mapped to a set s_C of n -arity tuples or to an n -arity predicate p_C . Quantifiers over relational objects of class C can be converted into quantifiers over either elements of s_C or tuples that satisfy p_C .

The remaining five rows in Table III show the results of different FOL encodings with quantifiers for the same set of benchmarks as in row 0. Rows 1 and 2 show the performance with z3 and cvc5, respectively, when encoding sets as predicates. z3 uses model-based quantifier instantiation (mbqi) by default and so we enabled it for cvc5 [15] as well. In both solvers, this technique is usually combined with E-matching to construct an interpretation that satisfies all generated ground instances of the input formula. This interpretation is either proved to satisfy the input formula as well or used to generate more ground instances that rule it out. z3 managed to solve all unsat benchmarks, and was overall superior to cvc5. However, it struggled with sat benchmarks. Rows 3 and 4 correspond to an encoding, with standard quantifiers, in cvc5’s theory of finite sets and relations. Row 3 shows the performance of enumerative instantiation [24], a technique that generates new instances based on a fair ordering of ground terms for the quantified variables. In contrast, row 4 shows the performance of finite model finding (fmf) [26], [25], which essentially enumerates possible interpretations. The two are complementary with respect to sat and unsat benchmarks solved. The last row shows the performance of set-bounded quantifiers using the filter operator σ as explained in Section IV. This approach is the clear winner on sat benchmarks among the direct encodings to SMT, and is competitive on unsat benchmarks. However, it solved 35 fewer benchmarks than LEGOS. Upon further investigation, we found that the solver keeps generating new element terms because of the presence of nested and alternating set.all and set.some quantifiers. Although this falls outside the decidable

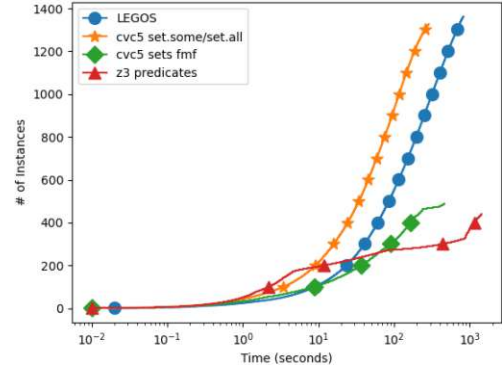


Fig. 3. Cactus plot showing the number of instances solved by each technique.

fragment, this limitation occurs in realistic benchmarks, so we plan to address it in future research.

In general, the results imply that using set-bounded quantifiers is more effective than using standard quantifiers in sat benchmarks. Figure 3 shows the number of benchmarks solved by each technique over time. Note how although the bounded-quantifier approach solved slightly fewer benchmarks than LEGOS, it was noticeably faster on the rest. This is possibly due to the overhead of the abstraction refinement loop in LEGOS, which generally requires multiple calls to the SMT solver, in contrast with the single call of our approach.

All experiments were run on a Linux machine with a 16-Core AMD EPYC 7313 processor. We used z3 version v4.13.4 and cvc5 version 1.2.0 (commit f3fc80e). All benchmarks used in this evaluation are publicly available [19].

It is worth mentioning that developing the encodings to cvc5 for SLEEC revealed a few soundness bugs in the SLEEC-to-FOL* encoding. For example, an improper handling of name collisions in quantified formulas was identified through discrepancies in satisfiability results with cvc5. This led to multiple rounds of bug fixes in both SLEEC and cvc5.

VI. CONCLUSION AND FUTURE WORK

We proved that the satisfiability of quantifier-free formulas in the theory of finite relations with filter and cross product operators is decidable when the satisfiability of quantifier-free formulas in the element theory is decidable and no set terms are used in filter predicates. We also showed how allowing set terms in filter predicates can compromise decidability. We demonstrated how to express set-bounded quantifiers using filter terms. Our experimental evaluation provided initial evidence that they are overall superior to standard quantifiers for SMT problems generated from a real application, and a viable alternative to a custom-made solver for such problems.

Our calculus is incomplete when we add the cardinality operator along with its rules [1]. We leave addressing this problem to future work. Multiset-bounded quantifiers were not discussed in this paper. However, similarly to set quantifiers, they could be easily implemented using the filter operator for multisets in [21]. This is also left to future work.

REFERENCES

- [1] Bansal, K., Barrett, C.W., Reynolds, A., Tinelli, C.: Reasoning with finite sets and cardinality constraints in SMT. *Log. Methods Comput. Sci.* **14**(4) (2018). [https://doi.org/10.23638/LMCS-14\(4:12\)2018](https://doi.org/10.23638/LMCS-14(4:12)2018), [https://doi.org/10.23638/LMCS-14\(4:12\)2018](https://doi.org/10.23638/LMCS-14(4:12)2018)
- [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: TACAS (1). *Lecture Notes in Computer Science*, vol. 13243, pp. 415–442. Springer (2022)
- [3] Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of recursive data types. *Electronic notes in theoretical computer science* **174**(8), 23–37 (2007). <https://doi.org/https://doi.org/10.1016/j.entcs.2006.11.037>
- [4] Cantone, D., Cutello, V., Policriti, A.: Set-theoretic reductions of hilbert's tenth problem. In: Börger, E., Büning, H.K., Richter, M.M. (eds.) *CSL '89*. pp. 65–75. Springer Berlin Heidelberg, Berlin, Heidelberg (1990), https://link.springer.com/chapter/10.1007/3-540-52753-2_32
- [5] Cantone, D., Cutello, V., Policriti, A.: Set-theoretic reductions of Hilbert's tenth problem. In: Börger, E., Büning, H.K., Richter, M.M. (eds.) *CSL '89*. pp. 65–75. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)
- [6] Cantone, D.: A survey of computable set theory. *Matematiche* **43**(1,2), 125–194 (1988), <https://lemaematiche.dmi.unict.it/index.php/lemaematiche/article/view/712>
- [7] Cantone, D., Ursino, P.: Decidability of the satisfiability problem for Boolean set theory with the unordered Cartesian product operator. *ACM Trans. Comput. Logic* **25**(1) (Jan 2024). <https://doi.org/10.1145/3626823>, <https://doi.org/10.1145/3626823>
- [8] Cantone, D., Zarba, C.G.: A tableau-based decision procedure for a fragment of set theory involving a restricted form of quantification. In: Murray, N.V. (ed.) *Automated Reasoning with Analytic Tableaux and Related Methods*. pp. 97–112. Springer Berlin Heidelberg, Berlin, Heidelberg (1999), https://link.springer.com/chapter/10.1007/3-540-48754-9_12
- [9] Cristiá, M., Rossi, G.: A practical decision procedure for quantifier-free, decidable languages extended with restricted quantifiers. *J. Autom. Reason.* **68**(4) (Oct 2024), <https://doi.org/10.1007/s10817-024-09713-6>
- [10] Cutler, J., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Ioannidis, E., Kastner, J., Mamat, A., McAdams, D., McCutchen, M., Rungta, N., Torlak, E., Wells, A.: Cedar: A new language for expressive, fast, safe, and analyzable authorization. In: *OOPSLA 2024* (2024), <https://www.amazon.science/publications/cedar-a-new-language-for-expressive-fast-safe-and-analyzable-authorization>
- [11] Feng, N., Marso, L., Sabetzadeh, M., Chechik, M.: Early verification of legal compliance via bounded satisfiability checking. In: Enea, C., Lal, A. (eds.) *Computer Aided Verification*. pp. 374–396. Springer Nature Switzerland, Cham (2023), https://link.springer.com/chapter/10.1007/978-3-031-37709-9_18
- [12] Feng, N., Marso, L., Yaman, S.G., Baartogtokh, Y., Ayad, R., de Mello, V.O., Townsend, B.A., Standen, I., Stefanakos, I., Imrie, C., Rodrigues, G.N., Cavalcanti, A., Calinescu, R., Chechik, M.: Analyzing and debugging normative requirements via satisfiability checking. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. pp. 214:1–214:12. ACM (2024). <https://doi.org/10.1145/3597503.3639093>, <https://doi.org/10.1145/3597503.3639093>
- [13] Feng, N., Marso, L., Yaman, S.G., Townsend, B., Cavalcanti, A., Calinescu, R., Chechik, M.: Towards a formal framework for normative requirements elicitation. In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. p. 1776–1780. ASE '23, IEEE Press (2024), <https://doi.org/10.1109/ASE56229.2023.00152>
- [14] Ferro, A., Omodeo, E.G., Schwartz, J.T.: Decision procedures for some fragments of set theory. In: Bibel, W., Kowalski, R. (eds.) *5th Conference on Automated Deduction Les Arcs, France, July 8–11, 1980*. pp. 88–96. Springer Berlin Heidelberg, Berlin, Heidelberg (1980), https://link.springer.com/chapter/10.1007/3-540-10009-1_8
- [15] Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009*. *Proceedings. Lecture Notes in Computer Science*, vol. 5643, pp. 306–320. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_25, https://doi.org/10.1007/978-3-642-02658-4_25
- [16] Jackson, D.: Software abstractions : logic, language, and analysis / daniel jackson. (2012), <https://mitpress.mit.edu/9780262528900/software-abstractions/>
- [17] Matijasevic, J.V.: Enumerable Sets are Diophantine, pp. 269–273. https://www.worldscientific.com/doi/abs/10.1142/9789812564894_0013
- [18] Meng, B., Reynolds, A., Tinelli, C., Barrett, C.W.: Relational constraint solving in SMT. In: de Moura, L. (ed.) *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science*, vol. 10395, pp. 148–165. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_10, https://doi.org/10.1007/978-3-319-63046-5_10
- [19] Mohamed, M., Feng, N.: Benchmarks for solving set constraints with comprehensions and bounded quantifiers (Jan 2025), <https://doi.org/10.5281/zenodo.14774207>
- [20] Mohamed, M., Feng, N., Reynolds, A., Tinelli, C., Barrett, C., Chechik, M.: Solving set constraints with comprehensions and bounded quantifiers (2025), <https://arxiv.org/abs/2508.08496>
- [21] Mohamed, M.M.Y., Reynolds, A., Tinelli, C., Barrett, C.: Verifying sql queries using theories of tables and relations. In: Björner, N., Heule, M., Voronkov, A. (eds.) *Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, EPIc Series in Computing*, vol. 100, pp. 445–463. EasyChair (2024). <https://doi.org/10.29007/rlt7>
- [22] de Moura, L., Björner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
- [23] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.* **1**(2), 245–257 (Oct 1979). <https://doi.org/10.1145/357073.357079>, <https://doi.org/10.1145/357073.357079>
- [24] Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 10806, pp. 112–131. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_7, https://doi.org/10.1007/978-3-319-89963-3_7
- [25] Reynolds, A., Tinelli, C., Goel, A., Krstic, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science*, vol. 8044, pp. 640–655. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_42, https://doi.org/10.1007/978-3-642-39799-8_42
- [26] Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., Barrett, C.W.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science*, vol. 7898, pp. 377–391. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_26, https://doi.org/10.1007/978-3-642-38574-2_26
- [27] Reynolds, A., Viswanathan, A., Barbosa, H., Tinelli, C., Barrett, C.W.: Datatypes with shared selectors. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings. Lecture Notes in Computer Science*, vol. 10900, pp. 591–608. Springer (2018), https://doi.org/10.1007/978-3-319-94205-6_39
- [28] Tinelli, C., Zarba, C.: Combining decision procedures for sorted theories. In: Alferes, J., Leite, J. (eds.) *Proceedings of the 9th European Conference on Logic in Artificial Intelligence (JELIA'04), Lisbon, Portugal. Lecture Notes in Artificial Intelligence*, vol. 3229, pp. 641–653. Springer (2004)

Learning Short Clauses via Conditional Autarkies

Amar Shah , Twain Byrnes , Joseph Reeves , Marijn J. H. Heule 

Carnegie Mellon University, Pittsburgh, PA, USA

{amarshah, binarynews, jereeves, marijn}@cmu.edu

Abstract—State-of-the-art Boolean satisfiability (SAT) solvers increasingly use techniques beyond resolution. One of the strongest such techniques is Propagation Redundant (PR) clause learning. Solvers utilizing PR clause learning may admit short proofs for benchmark families with exponentially large resolution proofs, including pigeonhole and mutilated chessboard. However, existing PR clause learning techniques require an NP-hard check; hence, they are computationally expensive and difficult to add to existing tools.

We propose a new technique for learning PR clauses based on conditional autarkies and implement it in the SAT solver CADICAL. Our method is modular, allowing for cross-solver compatibility, and learns PR clauses in linear time. Additionally, we introduce a number of heuristics, including a clause-shrinking technique and filtering to avoid trivial PR clauses, ensuring that our method learns useful clauses. We show that this is competitive with state-of-the-art PR clause learning techniques, and improves performance on a portion of SAT competition benchmarks.

I. INTRODUCTION

Boolean satisfiability (SAT) solving is a core tool in computer science with applications in program verification [1–4], planning [5, 6], cryptography [7], and mathematics [8–11]. As its usage expands, so too does the need for more powerful and specialized solving techniques.

One such class of techniques is propagation redundant (PR) clause learning [12]. In this technique, a solver generates and adds clauses which are satisfiability-preserving; that is, clauses whose conjunction with the formula is satisfiable if and only if the original formula is satisfiable. The solver aims to learn clauses that drastically shrink the potential solution space, such as clauses that break symmetries.

The power of this technique is tied to the strength of the underlying proof system: propagation redundancy (PR). A more powerful proof system can admit shorter proofs, which can be found and checked faster. Most SAT solvers rely on resolution-based proof systems, which are ineffective for many hard instances. Consider the pigeonhole principle (PHP), which asks if it is possible to fit $n + 1$ pigeons into n holes with at most one pigeon per hole. Using PR reasoning, one may learn a PR clause equivalent to the lemma: *pigeon 1 is not in hole 1*. Adding this clause restricts the search space for the first pigeon (and thus the problem), but it preserves satisfiability, as hole 1 is symmetric with all holes.

It is difficult to perform similar reasoning compactly with resolution, and in fact, PHP requires exponentially large resolution proofs [13]. The PR proof system, however, has short proofs for such problems, including a cubic proof (in the number of pigeons) for PHP [12]. PHP frequently occurs as a

subproblem for SAT benchmarks, so it is important to be able to solve it efficiently.

The theoretical power of PR learning comes at a cost: efficiently learning useful PR clauses is a challenge. State-of-the-art techniques such as Satisfiability Driven Clause Learning (SDCL) rely on calling another SAT solver to verify that a candidate clause is PR [14]. Thus, in the worst case, the solver takes exponential time to learn a single PR clause. This makes integration into high-performance solvers difficult and limits their practical impact. To the best of our knowledge, none of the most popular SAT solvers such as CADICAL [15], KISSAT [16], CRYPTOMINISAT [7], or LINGELING [17] support PR clause learning in their main branch.

In this work, we propose a new, efficient approach to learn PR clauses based on conditional autarkies. Intuitively, a conditional autarky provides a way to force the value of certain variables given a set of assumptions, e.g., in PHP if pigeons 3 to $n + 1$ are not in hole 1 or hole 2 (see Figure 1c for a visual representation), then pigeon 1 can be placed in hole 1 and pigeon 2 can be placed in hole 2. Kiesl et. al. proposed an algorithm for finding conditional autarkies in linear time [18], using them to delete clauses. In our work, we will use conditional autarkies to construct and add PR clauses.

While conditional autarkies present a means for bridging the theoretical gap in identifying PR clauses, the PR clauses they produce are often not immediately useful in real SAT solving applications. There are two major practical limitations: (1) larger PR clauses may not meaningfully constrain the search space, and (2) some smaller PR clauses may be trivial and distract the solver. We solve (1) by introducing a shrinking procedure to extract compact, useful PR clauses, and (2) by introducing a number of heuristics for filtering away trivial PR clauses. Returning to PHP, instead of the conditional autarky described above which produces a clause with at least $2n$ literals, it is possible through shrinking to learn a binary PR clause either forbidding pigeon 1 in hole 2 or pigeon 2 in hole 1.

We make the following contributions:

- 1) We introduce a method for learning PR clauses in linear time relative to the size of the formula.
- 2) We develop a number of shrinking and filtering heuristics to learn concise and useful PR clauses.
- 3) We implement these techniques in a solver, CAUTICAL¹ (a fork of the state-of-the-art SAT solver CADICAL),

¹CAUTICAL’s code is available at <https://github.com/amarshah10/cautical>.

and evaluate it on both PHP benchmarks and a suite of benchmarks from the annual SAT competitions.

II. BACKGROUND

We begin with some SAT preliminaries. Variables $x_1, x_2, \dots$ take values *true* ($\top$) or *false* ($\perp$). A literal l is a variable x or its negation $\bar{x}$. A clause C is a disjunction of literals, e.g., $C = x_1 \vee \bar{x}_4 \vee x_6$. A clause may also be represented by the set of its literals: e.g., the prior clause $C = \{x_1, \bar{x}_4, x_6\}$. Hence, we denote that a literal l occurs in clause C via $l \in C$ and the negation of clause is the set of negations of its literals, $\bar{C} = \{\bar{x}_1, x_4, \bar{x}_6\}$. A conjunctive normal form (CNF) formula Γ is a conjunction of (disjunctive) clauses. In this paper, all formulas Γ are in CNF.

We use $\text{var}(\Gamma)$ to represent the set of variables occurring in formula Γ . An *assignment* $\alpha : V \rightarrow \{\top, \perp\}$ on Γ maps variables $V \subseteq \text{var}(\Gamma)$ to $\top$ or $\perp$. If $V = \text{var}(\Gamma)$, then α is a *total assignment*. An assignment that is not total is called a *partial assignment*. For this paper, assignment refers to both total and partial assignments. We abuse notation to extend an assignment α to literals, by denoting $\alpha(l) = \alpha(x)$ if $l = x$ and $\alpha(l) = \neg\alpha(x)$ if $l = \bar{x}$.

A formula restricted to an assignment $\Gamma|_\alpha$ is the formula resulting from mapping variables in the domain of α to their assignments. We can simplify such a formula by removing literals assigned $\perp$ and clauses containing a literal assigned $\top$ (which are thus satisfied). For example, with formula $\Gamma = (x_1 \vee \bar{x}_4 \vee x_6) \wedge (x_2 \vee x_3) \wedge \bar{x}_5$ and assignment α mapping x_1 to $\perp$ and x_2 to $\top$, we have $\Gamma|_\alpha = (\bar{x}_4 \vee x_6) \wedge \bar{x}_5$.

A clause $C = l_1 \vee \dots \vee l_m$ *blocks* an assignment mapping each $l_i \in C$ to false ($\perp$). An assignment α *touches* a clause C if there is a variable x assigned by α such that $x \in C$ or $\bar{x} \in C$. We say α *satisfies* C if there is some $l_i \in C$ with $\alpha(l_i) = \top$.

Assignment α *satisfies* formula Γ if it satisfies every clause $C \in \Gamma$. If such a satisfying assignment exists, then Γ is *satisfiable*.

Redundant Clauses: A clause C is *redundant* (or *satisfiability-preserving*) with respect to a formula Γ if the formulas Γ and $\Gamma \wedge C$ are *equisatisfiable*, i.e., Γ is satisfiable if and only if $\Gamma \wedge C$ is satisfiable.

In a *clausal proof system*, each step adds or removes a redundant clause C . The step may contain extra information, such as a boolean witness justifying C 's redundancy. A list of redundant clauses ending with the empty clause $\perp$ is a proof of unsatisfiability for formula Γ .

Adding a redundant clause may help a solver by greatly constraining the set of possible solutions, which gives the solver a smaller solution space to search; however, the addition of these clauses could also negatively interact with solver heuristics, increasing solve time.

Clausal proof systems range in complexity, with one of the simplest derivation rules being resolution. Given two clauses, $C \vee x$ and $\bar{x} \vee D$, *resolution* produces the logically implied clause $C \vee D$.

Unit propagation is a core reasoning technique in a SAT solver. Starting with empty assignment α , if a formula Γ contains a *unit clause* l , i.e., a clause with only a single literal, we set $\alpha(l) = \top$. Then, this unit is propagated: we consider the unit clauses of the formula $\Gamma|_\alpha$ and continue this process until no unit clauses remain. If unit propagation terminates with $\Gamma|_\alpha = \emptyset$, we say that it derives a conflict.

Given a formula Γ and clause $C = l_1 \vee \dots \vee l_k$, we can say $\Gamma \vdash_1 C$, read as “ Γ implies C via unit propagation,” if $\Gamma \wedge \bar{C} \equiv \Gamma \wedge \bar{l}_1 \wedge \dots \wedge \bar{l}_k$ derives a conflict after applying unit propagation. Here, C is a *reverse unit propagation (RUP)* clause, a simple example of a redundant clause with respect to Γ . For some formula Γ' , we say $\Gamma \vdash_1 \Gamma'$ if $\Gamma \vdash_1 C$ for every clause $C \in \Gamma'$.

Definition 1. [*Propagation Redundant (PR) clauses* [12]] For formula Γ , clause C , and assignment α blocked by C , we say that C is *propagation redundant (PR)* if there exists a witness assignment ω such that ω satisfies C and

$$\Gamma|_\alpha \vdash_1 \Gamma|_\omega.$$

Informally, a clause C is PR if every assignment that satisfies Γ but falsifies C can be turned into an assignment that satisfies $\Gamma \wedge C$. While this property can be validated in polynomial time (by unit propagation), checking if a clause is PR without hints is NP-complete [12]. Thus, the propagation redundancy (PR) proof system introduces witnesses, which must be provided for proof checking.

PR clauses subsume many classes of redundant clauses, including resolution asymmetric tautologies (RATs) [19], blocked clauses [20], set-blocked clauses [21], and globally-blocked clauses [18].

A. Conditional Autarkies

Definition 2 (Autarky [18]). A nonempty assignment α is an *autarky* for a formula Γ if every clause $C \in \Gamma$ touched by α is satisfied.

Simply, an autarky is an assignment that satisfies every clause it touches. Unfortunately, formulas often do not contain autarkies, and they are difficult to find even when they exist. Instead, we use the following weakening of an autarky:

Definition 3 (Conditional Autarky [18]). A nonempty assignment $\alpha = \alpha_c \sqcup \alpha_a$ (disjoint union) is a *conditional autarky* for a formula Γ if α_a is an autarky for $\Gamma|_{\alpha_c}$.

Consider, as an example, the formula with aptly named variables $\Gamma = (c \vee \bar{a}) \wedge (a \vee x) \wedge (\bar{c} \vee \bar{y})$. We have a conditional autarky $\alpha = \alpha_c \sqcup \alpha_a$, with $\alpha_c = c$ and $\alpha_a = a$. Since $\Gamma|_{\alpha_c} = (a \vee x) \wedge (\bar{y})$ and a occurs here only positively, a is an autarky for $\Gamma|_{\alpha_c}$. Thus, α is a conditional autarky.

Conditional autarkies can be very useful for learning satisfiability-preserving clauses as in the following theorem from Kiesl et al. [18]:

Theorem 1. Let Γ be a formula and $\alpha = \alpha_c \sqcup \alpha_a$ be a conditional autarky on Γ , with $\alpha_c = c_1, \dots, c_n$ and $\alpha_a = a_1, \dots, a_m$. Formula Γ and $\Gamma \wedge \bigwedge_{1 \leq i \leq m} (\bar{c}_1 \vee \dots \vee \bar{c}_n \vee a_i)$ are equisatisfiable.

A solver may thus add any of the m clauses $\bar{c}_1 \vee \dots \vee \bar{c}_n \vee a_i$ and preserve satisfiability.

Intuitively, Theorem 1 states that adding this implication preserves satisfiability:

$$[c_1 \wedge \dots \wedge c_n] \rightarrow [a_1 \wedge \dots \wedge a_m]$$

When converted to clausal form, this results in the m different clauses from the theorem. Indeed, each of these is a PR clause with witness α_a .

B. Related Work

Solvers that implement the PR proof system typically use the satisfaction-driven clause learning (SDCL) framework [22], which extends conflict-driven clause learning (CDCL) [23]. After propagating an assignment, they check if the clause C blocking this assignment is PR by creating a new SAT formula called the *positive reduct*. If the positive reduct is satisfiable, then C is a PR clause and is added. This was implemented in an extension of the solver LINGELING and was shown to scale well on pigeonhole benchmarks.

Later, two new variants of the positive reduct were proposed for more aggressive pruning of the search space [14]. This allowed SDCL to solve other difficult problems such as mutilated chessboard [24] and Tseitin formulas over expander graphs [25]. This is implemented in a new SDCL solver SADICAL.

PRELEARN is a preprocessing technique for PR clauses [26]. It initially considers many possible clauses and queries SADICAL to see which are PR.

Our work differs as we do not use a *positive reduct* to test if a clause is PR. Instead, our clauses are PR by construction, as they come from a conditional autarky. This has the potential downside that our clause may be large, and thus weak. To remedy this, we apply a shrinking technique, reducing the size of a clause. Additionally, prior techniques are sensitive to the encoding of the problem. Minor changes such as literal and clause reordering can have a large effect on performance. We compare our implementation CAUTICAL to SADICAL and PRELEARN in Section V.

Kiesl et al. [18] first introduced conditional autarkies to identify a class of PR clauses known as globally blocked clauses. They aimed to eliminate globally blocked clauses from a formula to simulate circuit-simplification techniques. Our work adds clauses instead of removing them.

III. METHODOLOGY

A. Motivating Example

We use the pigeonhole principle as a motivating example. The problem PHP(n) asks whether we can put $n + 1$ pigeons in n holes such that (1) every pigeon is in a hole, and (2) no hole contains more than one pigeon. This can be encoded as a SAT problem where variable $x_{i,j}$ represents putting the i -th pigeon into the j -th hole. Constraint (1) is encoded as $\bigvee_{1 \leq j \leq n} x_{i,j}$ for each $1 \leq i \leq n + 1$ and (2) as $\bar{x}_{i,j} \vee \bar{x}_{k,j}$ for each $1 \leq i < k \leq n + 1$ and $1 \leq j \leq n$.

Figure 1 provides a visualization where the rows represent the pigeons and the columns represent the holes. The cell in row i and column j represents the literal $x_{i,j}$. A $+$ in the (i, j) -th cell indicates that $x_{i,j}$ is set to $\top$, and a $-$ symbol indicates $\perp$. Thus, constraint (1) asks that each row has at least one $+$ and constraint (2) asks that each column has at most one $+$.

We achieve an $O(n^3)$ PR proof for PHP(n), matching the best known result [12]. In Figure 1 we learn the clause $\bar{x}_{1,2} \vee \bar{x}_{2,1}$. After learning n such clauses, we learn the clause $\bar{x}_{1,2}$, i.e. “pigeon 1 is not in hole 2.” Since there are $n+1$ pigeons and n holes, we must rule out $O(n^2)$ pigeon-hole pairs, so the proof has size $O(n^3)$. We learn these proofs with a very low constant factor and robustness against encoding perturbations (see Subsection V-A).

B. PR Clause Learning Framework

Algorithm 1: Learning PR clauses

```

1 Function LearnClause( $\Gamma, \alpha$ ):
2   for  $i \in \text{vars}(\Gamma)$  :
3     for  $j \in \text{vars}(\Gamma)$  :
4       Propagate( $i$ );
5       Propagate( $j$ );
6        $\alpha_c, \alpha_a := \text{LeastConditional}(\Gamma, \alpha)$ ;
7        $C := \text{Shrink}(\Gamma, \alpha_c, \alpha_a)$ ;
8       if not Filter( $C, \Gamma$ ) :
9          $\Gamma := \Gamma \wedge C$ ;
10      Backtrack();
```

We present the general framework for using conditional autarkies to learn PR clauses in Algorithm 1. A conditional autarky is computed based on an assignment, so the algorithm first selects a set of literals to propagate (lines 4, 5), creating the assignment α . We propagate two literals at a time using nested **for** loops, and undo the propagations after each iteration (backtracking in line 10). In line 6 Algorithm 2 makes a single pass over the formula to generate a conditional autarky, with conditional part α_c and autarky part α_a . The PR clause C is generated from the conditional autarky and shrunk in line 7, and if C does not pass the usefulness heuristics it is filtered away in line 8. Details regarding design choices and heuristics are found in Section IV, including a discussion of Filter.

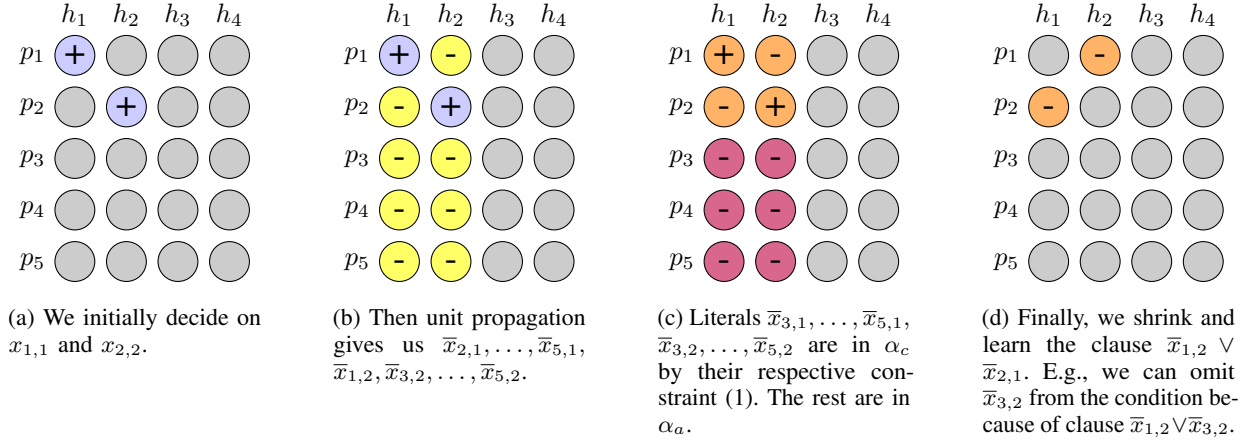


Fig. 1: Learning the clause $\bar{x}_{1,2} \vee \bar{x}_{2,1}$ for PHP(4)

In the following sections, we will describe PR clause learning and shrinking in the context of our running example PHP(4). We will consider the decisions $i = x_{1,1}$ and $j = x_{2,2}$ shown in Figure 1a, with unit propagation shown in Figure 1b.

C. Learning PR Clauses

As described in Subsection II-A, any assignment can be split into a conditional and a (potentially empty) autarky part, $\alpha = \alpha_c \sqcup \alpha_a$, and the following clauses are PR: $\bigvee_{c \in \alpha_c} \bar{c} \vee a$ for any $a \in \alpha_a$. In order to produce smaller PR clauses, we use Algorithm 2 from Kiesel et al. [18] to find the unique smallest possible α_c .

Algorithm 2: Unique minimal α_c in $\alpha = \alpha_c \sqcup \alpha_a$

```

1 Function LeastConditional( $\Gamma, \alpha$ ):
2    $\alpha_c := \emptyset$ ;
3   for  $C \in \Gamma$  :
4     if  $\alpha$  touches  $C$  without satisfying  $C$  :
5        $\alpha_c := \alpha_c \cup (\alpha \cap \bar{C})$ ;
6   return  $\alpha_c, \alpha \setminus \alpha_c$ ;
```

Algorithm 2 returns $\alpha_c \sqcup \alpha_a$, which is a conditional autarky, as every clause that α_a touches is satisfied by a literal in α .

Additionally, α_c is minimal: for any other conditional autarky $\alpha = \alpha'_c \sqcup \alpha'_a$, it must be the case that $\alpha_c \subseteq \alpha'_c$ since for each clause that is touched but not satisfied, we add to α_c all literals from the assignment that touch and do not satisfy this clause. These literals must be in α'_c , since otherwise α'_a will touch a clause that is not satisfied by α , violating the conditional autarky property.

Running Algorithm 2 on the assignment from Figure 1b gives the conditional part (red) and autarky part (orange) in Figure 1c. The assigned literals from pigeons 3, 4, and 5 appear in α_c because they touch but do not satisfy the constraint (1) clauses for the respective pigeons stating that every pigeon is

in a hole. However, the constraint (1) clauses are satisfied for pigeons 1 and 2, along with the touched constraint (2) clauses, placing the pigeons 1 and 2 literals in α_a . Intuitively, if pigeons 3, 4, and 5 are not in holes 1 or 2, then pigeon 1 can be placed in hole 1: $x_{3,1} \vee x_{3,2} \vee x_{4,1} \vee x_{4,2} \vee x_{5,1} \vee x_{5,2} \vee x_{1,1}$ and pigeon 1 can be kept out of hole 2: $x_{3,1} \vee x_{3,2} \vee x_{4,1} \vee x_{4,2} \vee x_{5,1} \vee x_{5,2} \vee \bar{x}_{1,2}$ (likewise for pigeon 2 but with the holes swapped). The shrinking technique in the following section will help reduce the size of these large PR clauses.

D. Shrinking PR Clauses

The PR clause derived from a conditional autarky can be too weak to effectively reduce the search space. Shrinking the PR clause, i.e., removing literals from the clause via resolution, will strengthen its pruning power.

At a high-level, we will generate two sets: $C_0 \subseteq \alpha_c$ and $A_0 \subseteq \alpha_a$, and show that we can learn the PR clause $\bigvee_{c \in C_0} \bar{c} \vee \bigvee_{a \in A_0} a$.

First we start with C_0 , the set of literals in the conditional part that are inconsistent with any literal in the autarky part. Two literals l_i and l_j are inconsistent in a formula Γ if $\Gamma \wedge l_i \vdash \bar{l}_j$, meaning the clause $\bar{l}_i \vee \bar{l}_j$ is RUP. Formally, $C_0 = \{c_j \in \alpha_c \mid \exists a_i \in \alpha_a \text{ s.t. } \Gamma \wedge \bar{a}_i \vdash c_j\}$, with a_i appearing negated in the inconsistency check so that we can perform a resolution between the derived binary clause $a_i \vee c_j$ and the original PR clause C .

Next we generate A_0 which is a subset of the autarky literals that are together inconsistent with the literals in C_0 . So, for each $c \in C_0$ there exists an $a \in A_0$ such that $\Gamma \wedge \bar{a} \vdash c$. There always exists at least one A_0 , namely α_a which is used to define C_0 , but if a smaller A_0 exists it can be used to shrink the size of the PR clause.

Theorem 2. *Let Γ be a formula and $\alpha = \alpha_c \sqcup \alpha_a$ be a conditional autarky on Γ , with $\alpha_c = c_1, \dots, c_n$ and $\alpha_a = a_1, \dots, a_m$. Let $A_0 \subseteq \alpha_a$ be non-empty. Let $C_0 = \{c_j \in \alpha_c \mid$*

$\exists a_i \in A_0$ s.t. $\Gamma \wedge \bar{a}_i \vdash_1 c_j$. Then formula Γ is satisfiable if and only if $\Gamma \wedge (\bigvee_{c \in C \setminus C_0} \bar{c} \vee \bigvee_{a \in A_0} a)$ is satisfiable.

Proof. $\Leftarrow$: This is immediate

$\Rightarrow$: From Theorem 1, Γ is satisfiable implies $\Gamma \wedge (\bigvee_{c \in C} \bar{c} \vee a)$ is satisfiable for any $a \in \alpha_a$. We will pick an $a \in A_0$. Hence, $\Gamma \wedge (\bigvee_{c \in C} \bar{c} \vee \bigvee_{a \in A_0} a)$ is satisfiable since we are weakening a clause.

By the definition of inconsistency, for each $c \in C_0$ there exists an $a \in A_0$ such that $\Gamma \wedge \bar{a} \vdash_1 c$, allowing us to derive the RUP clause $a \vee c$. Each binary clause can be resolved with $(\bigvee_{c \in C} \bar{c} \vee \bigvee_{a \in A_0} a)$, producing $(\bigvee_{c \in C \setminus C_0} \bar{c} \vee \bigvee_{a \in A_0} a)$. $\square$

In fact, $\bigvee_{c \in C \setminus C_0} \bar{c} \vee \bigvee_{a \in A_0} a$ is a PR clause with witness α_a .

Greedy Set Cover. We can interpret the problem of finding the smallest possible A_0 as a set cover problem where each literal $a \in \alpha_a$ defines the set $SETS(a) = \{c \in \alpha_c \mid \Gamma \wedge \bar{a} \vdash_1 c\}$. Finding the minimum set cover is NP-hard, so we instead approximate using a greedy algorithm, returning a set cover at most roughly $(\ln |\alpha_c| + 1) \times$ the size of the smallest set cover [27].

The greedy algorithm initializes C_0 as empty, and in each iteration finds the $a \in \alpha_a$ that generates the largest set $SETS(a) \cap (\alpha_c \setminus C_0)$, adding a to A_0 and adding $SETS(a)$ to C_0 . The algorithm terminates once all sets in $\{SETS(a) \cap (\alpha_c \setminus C_0) \mid a \in \alpha_a\}$ are empty. Notice that oftentimes, $C_0 \subsetneq \alpha_c$, i.e. we do not achieve a complete cover of α_c .

Algorithm 3: Algorithm finding A_0

```

1 Function Shrink( $\Gamma, \alpha_a, \alpha_c$ ):
2    $SETS := \text{init Array}[\text{len}(\alpha_a)]$ ;
3   for  $i \in \text{range}(\alpha_a)$  :
4     Propagate( $\alpha_a[i]$ );
5      $\text{implied} := \{\}$ ;
6     for  $c \in \alpha_c$  :
7       Propagate( $\bar{c}$ );
8       if  $\text{unsat}$  :
9          $\text{implied} := \text{implied} \cup \{c\}$ ;
10      Backtrack(1);
11       $SETS[i] := \text{implied}$ ;
12 return GreedySetCover( $SETS$ );
```

In Algorithm 3, we describe our process for calculating A_0 . We initialize $SETS$ as an array (line 2), iterate the counter i through α_a (line 3), populating $SETS[i]$ with the set of literals $c \in \alpha_c$ such that $\Gamma \wedge \bar{a}_i \vdash_1 c$ (lines 4-11). Finally, we apply GreedySetCover to get a small set A_0 that covers as much of C as possible.

Returning to the pigeonhole example, C_0 contains all of the literals in α_c because of the binary clauses in constraint (2). The smallest possible A_0 includes $\bar{x}_{2,1}$ and $\bar{x}_{1,2}$, whose sets cover $\bar{x}_{3,1}, \bar{x}_{4,1}, \dots, \bar{x}_{5,1}$ and $\bar{x}_{3,2}, \bar{x}_{4,2}, \dots, \bar{x}_{5,2}$ respectively. Whereas, the sets from $x_{1,1}$ and $x_{2,2}$ are empty (they are not

inconsistent with the other literals). Therefore, we can learn the clause $\bar{x}_{2,1} \vee \bar{x}_{1,2}$, shown in Figure 1d.

IV. IMPLEMENTATION

We implement our technique in CAUTICAL (a fork of CADICAL). We choose CADICAL as it has shown strong performance in the SAT Competition. For instance, a fork of CaDiCaL won in 2023 [28]. Our techniques can be implemented in any CDCL SAT solver, but we leave this as future work.

By default, CAUTICAL spends 30 seconds searching for PR clauses in a preprocessing step. After this time limit, the solver exits and commences normal solving, regardless of whether or not it has found any PR clauses. This adds ~ 800 lines of C++ code to CADICAL and is implementable within any CDCL SAT solver.

We discuss three important design decisions in CAUTICAL:

- 1) **shrink**: As discussed in Subsection III-D, we may shrink a clause using the techniques inspired by greedy set cover.
- 2) **filter-triv**: We filter out clauses that are trivial, i.e. if $\Gamma \vdash_1 C$ for some clause C .
- 3) **filter-long**: We filter out clauses that are longer than some set length l . We pick $l = 2$ as a default and thus only learn binary and unit clauses.

The filter function in Algorithm 1 combines **filter-triv** and **filter-long**, disallowing trivial clauses or clauses with length greater than 2.

Additionally, we describe three natural design decisions that we did not make:

- 1) **longer-preprocess**: We choose 30 seconds as the default preprocessing time. However, we could choose a longer time limit, for instance 100 seconds.
- 2) **order-i**: On Algorithm 1 line 2, by default we pick i randomly. However, we could choose i based on some ordering. For instance, we could order literals i by how frequently they occur in the original formula.
- 3) **select-j**: On Algorithm 1 line 3, we iterate through all possible literals j . However, we could choose some subset of possible literals j . One such example is to pick j from the neighbors of i . This is implemented in PRELEARN. The neighbors of i include any literal j that belongs to the same clause as a literal fixed by unit propagation on i (including i itself).

We evaluate these heuristics in Subsection V-D and find that **shrink**, **filter-triv**, and **filter-long** are all beneficial. On the other hand **longer-preprocess**, **order-i**, and **select-j** are not beneficial and can sometimes be harmful.

V. EVALUATION

In this section, we empirically evaluate our technique against other PR clause learning techniques. In doing so, we aim to answer the following research questions:

- RQ1 Can our approach provide a speedup on certain benchmark families?

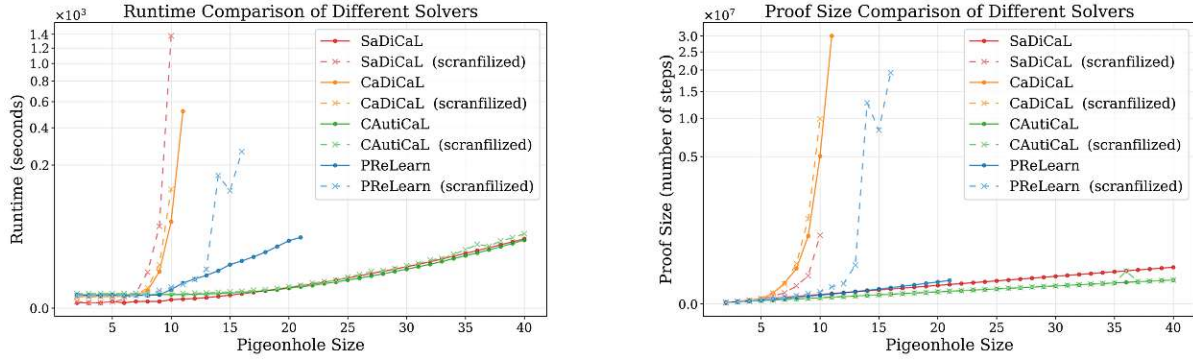


Fig. 2: Comparison of CAUTICAL, CADICAL, SADICAL, and PRELEARN on pigeonhole principle benchmarks up to size 40. The y-axis is on a cube root scale. The performance of a solver on the original benchmark is shown with a solid line. The median of 5 scanfilized queries is shown with a dashed line. If a solver times out on a query in 5000s, it is not shown.

RQ2 Is our approach less sensitive to encoding choices compared to other PR learning techniques?

We compare the two main tools learning PR clauses: SADICAL (based on SDCL) and PRELEARN (a preprocessing technique that calls SADICAL). To be consistent with our approach, we run PRELEARN with its default settings for 30 seconds, then solve the preprocessed formula with CADICAL. We also compare to CADICAL as a baseline with no PR clause learning. We check all proofs of unsatisfiability from CAUTICAL using `dpr-trim` [29].

All experiments were performed in the Anvil Supercomputing Center on nodes with 128 cores and 2 GB RAM per core [30]. We ran 64 experiments in parallel per node with a 5,000 second timeout, the default timeout for the SAT competition.

In Subsection V-A, we compare all approaches on the pigeonhole principle, evaluating runtime, proof length, and sensitivity to the encoding of the formula. In Subsection V-B, we evaluate the solvers on benchmarks from the '22, '23, and '24 SAT competition's main tracks [28, 31, 32]. In Subsection V-C, we highlight certain benchmark families that benefit from PR clause learning. In Subsection V-D, we evaluate the benefit of different heuristic choices in CAUTICAL. Finally, we conclude in Subsection V-E with a discussion of how performed on RQ1 and RQ2.

A. Pigeonhole results

Approaches based on SDCL, such as SADICAL, are successful for learning $O(n^3)$ proofs for the pigeonhole principle, but are very sensitive to the encoding of the formula. We compare the solvers on pigeonhole principle from PHP(2) to PHP(40) and plot these results in Figure 2. As the expected best-behavior is cubic, we use a cube root scale for the y-axis.

As expected, CADICAL grows exponentially, while SADICAL and CAUTICAL scale cubically in both runtime and proof size. Significantly, CAUTICAL is able to learn $3.59\text{--}3.64\times$ shorter proofs compared to SADICAL. PRELEARN

scales cubically on small formulas, but for PHP(22) and larger, will not learn enough useful PR clauses in the preprocessing step and will timeout after spending the rest of its time running CADICAL.

Additionally, we evaluate all solvers on scanfilized variations of the pigeonhole principle. Scanfilization is a technique for generating an satisfiability-equivalent formula [33]. We use the tool `scanfilize` [33] with the options permuting variables, permuting clauses, and flipping literals (with probability 0.5) all turned on. We run each solver on 5 scanfilized variations for each benchmark and take the median runtime and proof size. This is shown in Figure 2 with dashed lines.

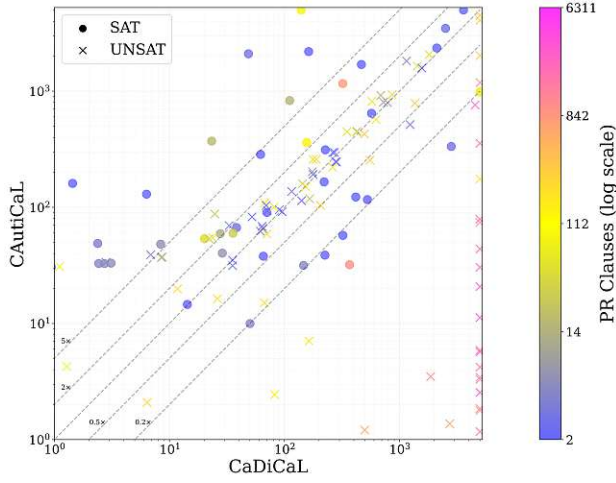
SADICAL and PRELEARN exhibit an exponential trend for runtime and proof size on the scanfilized benchmarks. SADICAL will spend all its time in the main SDCL loop not learning enough useful clauses. PRELEARN will learn some useful PR clauses in preprocessing, but not enough to sufficiently shrink the search space for formulas larger than PHP(16).

On the other hand, CAUTICAL almost matches its non-scanfilized performance, demonstrating that it learns useful PR clauses regardless of the encoding.

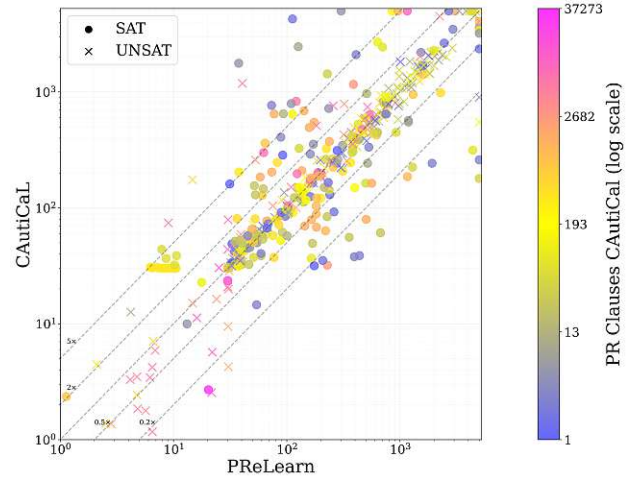
B. SAT competition results

We compare the performance of CAUTICAL with CADICAL and PRELEARN on the benchmarks from the '22, '23, and '24 SAT competition's main tracks [28, 31, 32]. We remove duplicates and exclude all benchmarks with more than twenty million clauses as these are out of scope for our technique. This gives us a total of 1,089 benchmarks. We exclude SADICAL from our evaluation as it only solves 22 of these benchmarks.

Table I shows the number of instances solved by each solver. Out of the total number solved, it shows the number of formulas for which PRELEARN and CAUTICAL learn additional PR clauses, improve upon CADICAL by at least 5%, and solve a formula that CADICAL does not solve.



(a) Comparing CAUTICAL to CADICAL.



(b) Comparing CAUTICAL to PRELEARN.

Fig. 3: Performance comparison of CAUTICAL with PRELEARN and CADICAL on SAT competition benchmarks. On each graph, we filter out all benchmarks where neither solver learns any PR clauses. The color indicates the number of PR clauses learnt by CAUTICAL.

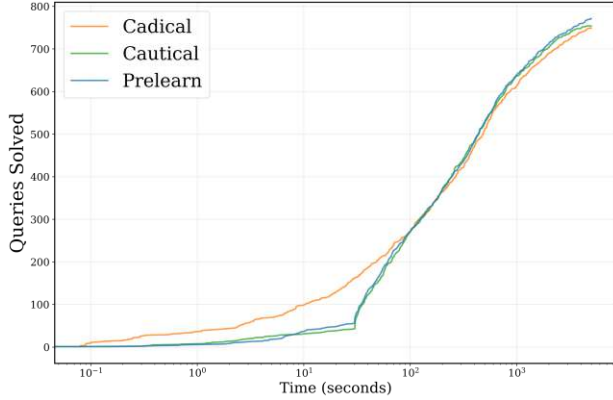


Fig. 4: The number of formulas solved by CAUTICAL, PRELEARN, and CADICAL.

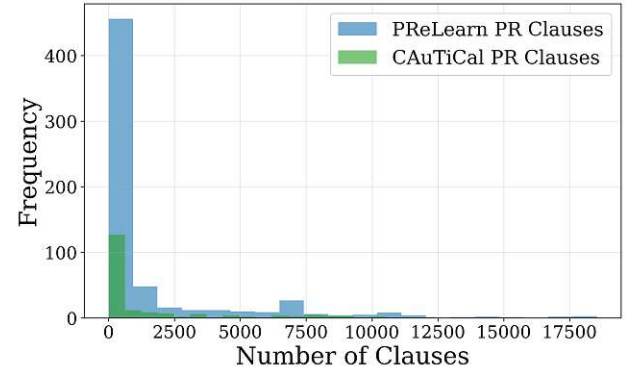


Fig. 5: The size of PR clauses learnt by CAUTICAL and PRELEARN

We divide the benchmarks based on number of clauses (0-10k or 10k-20M) and status (SAT or UNSAT). The number of clauses is a good indicator of the type of benchmark, with many hard combinatorial problems containing fewer than ten thousand clauses.

Figure 3 shows CAUTICAL and CADICAL’s performance relative to CADICAL. Figure 3a shows that CAUTICAL learns a large number of PR clauses for formulas which it solves quickly and CADICAL times out on. Figure 3b shows that PRELEARN also solves a number of formulas that CAUTICAL cannot, but CAUTICAL typically performs better on formulas where it learns many PR clauses.

We find that CAUTICAL can show a performance improvement or degradation on formulas where it does not learn any PR clauses. This is because as CAUTICAL runs, it updates internal data structures such as watched literals, clause occurrence lists, and variable phases.

For instance, during the initial phase of searching for PR

clauses, the solver will decide on a literal. Unit propagation of this literal may lead to a conflict, and the solver can learn the unit clause without any autarky/PR reasoning. Unit clauses are not stored or treated as learned clauses inside CADICAL, but instead the literal becomes “fixed,” i.e., the solver treats the literal as always true. For instance, this happens on the three satcoin benchmarks where CAUTICAL shows the most improvement (see Figure 6).

PAR-2 score is a standard metric used to evaluate the performance of solvers. It is evaluated as the sum of the runtimes of solved instances and twice the timeout of unsolved instances. On this dataset, CADICAL has a PAR-2 score of 3522 seconds, PRELEARN has a PAR-2 score of 3331 seconds, and CAUTICAL has a PAR-2 score of 3442 seconds.

Figure 4 shows the distribution of benchmarks solved by each solver. Both PRELEARN and CAUTICAL lag behind CADICAL during the 30 second preprocessing stage, but eventually solve

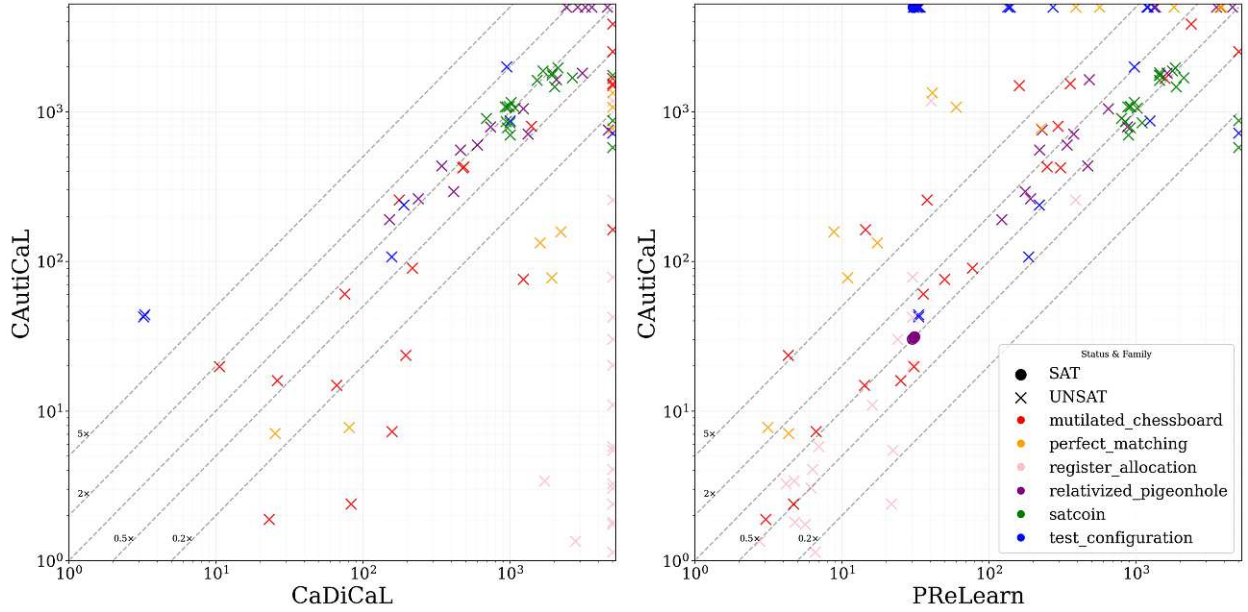


Fig. 6: Comparing CAUTICAL with CADICAL and PRELEARN on various benchmark families.

TABLE I: Number of formulas that CADICAL, PRELEARN and CAUTICAL solve, divided between 0-10k clause and 10k-20M clause formulas, and SAT and UNSAT formulas. For PRELEARN and CAUTICAL we include the number of benchmarks with PR clauses learnt, more than 50 PR clauses learnt, improved on CADICAL by 5% or more; and solved that CADICAL did not solve.

	0–10k clause		10k–20M clause		Total
	SAT	UNSAT	SAT	UNSAT	
CADICAL Solved	54	73	319	303	749
PRELEARN					
Total	52	90	322	307	771
Learn PR clause	40	73	179	145	431
Learn > 50 PR clauses	22	51	104	79	256
Improve on CADICAL	11	42	57	36	146
Unique from CADICAL	1	17	6	6	30
CAUTICAL					
Total	52	87	317	298	754
Learn PR clause	16	58	30	35	139
Learn >50 PR clauses	1	39	6	11	57
Improve on CADICAL	23	48	89	59	219
Unique from CADICAL	0	18	9	9	36

more benchmarks. In the end, PRELEARN solves the most formulas at 771, followed by CAUTICAL at 754, and CADICAL at 749.

CAUTICAL exercises more selective PR clause learning techniques, only learning PR clauses for 139 formulas, while PRELEARN learns PR clauses for 431 formulas (see Figure 5). On those formulas, CAUTICAL improves CADICAL’s runtime by 5% or more on 29.9% of benchmarks, while PRELEARN improves it on 24.7% of benchmarks.

C. Discussion of Benchmark Families

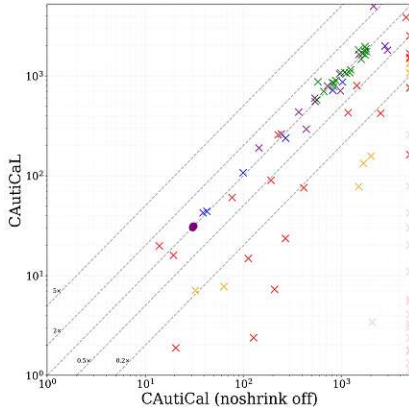
We identify six benchmark families for which PR clauses perform well. We choose them based on prior work [26] and our experiments on SAT competition benchmarks:

- 1) *mutilated-chessboard*: famous problem asking if one can use 2-by-1 tiles to cover an $2n$ -by- $2n$ chessboard with opposite corners removed. This is difficult for resolution [34], but there exists $O(n^3)$ PR proofs [24].
- 2) *perfect_matching*: generalization of the pigeonhole principle and mutilated chessboard problems with various at-most-one constraints [35].
- 3) *register_allocation*: the graph coloring problem generated by simulating register allocation on individual Python functions [36].
- 4) *relativized_pigeonhole*: generalization of the pigeonhole principle where we place $n + 1$ pigeons in n holes with k nesting places.
- 5) *satcoin*: variant of a bitcoin mining problem [37].
- 6) *test_configuration*: is there a list of configurations of size k that covers every pairwise combination of configurations of a SAT solver [38].

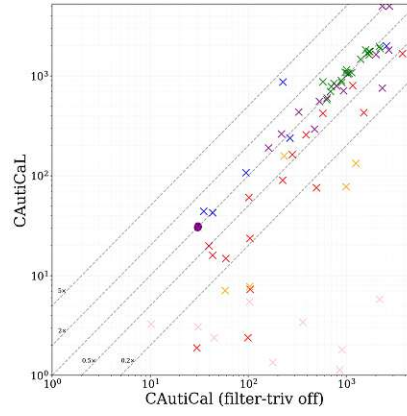
TABLE II: Overview of benchmark families used in evaluation

Benchmark Family	Number of Clauses	CAUTICAL # PR
mutilated-chessboard	900-3K	115-351
perfect_matching	300-1K	183-447
register_allocation	1K-23K	671-3322
relativized_pigeonhole	3K-2M	469-9091
satcoin	600K-600K	0-0
test_configuration	31K-64K	0-0

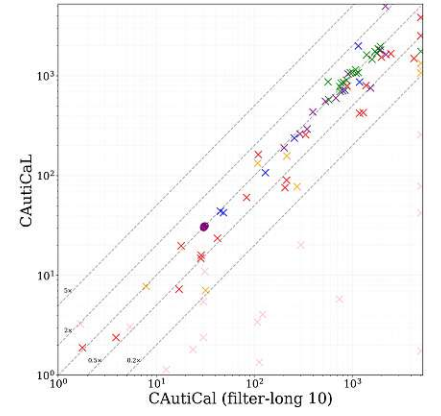
Figure 6 compares the performance of CAUTICAL with



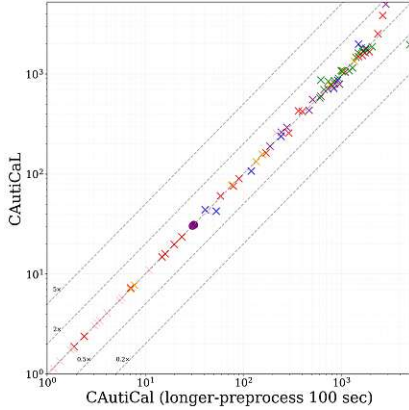
(a) CAUTICAL compared to CAUTICAL with shrink turned off



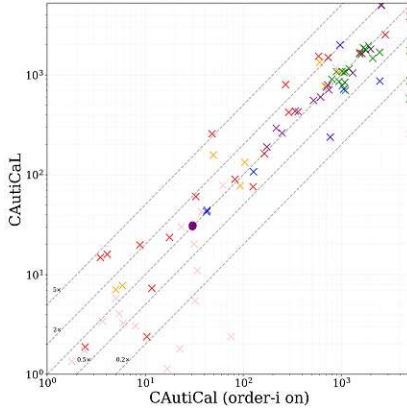
(b) CAUTICAL compared to CAUTICAL with filter-triv turned off



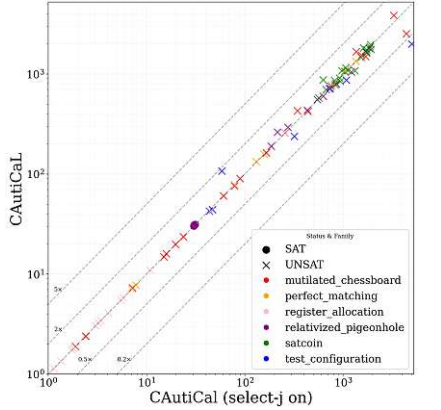
(c) CAUTICAL compared to CAUTICAL with filter-long set to 10



(d) CAUTICAL compared to CAUTICAL with longer-preprocess set to 100 seconds



(e) CAUTICAL compared to CAUTICAL with order-i turned on



(f) CAUTICAL compared to CAUTICAL with select-j turned on

Fig. 7: Performance comparison of CAUTICAL with various heuristics turned on and off

CADICAL and PRELEARN on these benchmark families. We do not evaluate SADICAL as it does very poorly on SAT competition benchmarks. Table II provides a brief overview of the families: the number of clauses in the original formulas (up to 1 significant digit) and the number of PR clauses learnt by CAUTICAL.

CAUTICAL compares favorably to CADICAL on all benchmark families, showing especially large speedups on `register_allocation`, `mutilated_chessboard` and `perfect_matching`.

PRELEARN exhibits large speedups over CAUTICAL on `test_configuration` and `perfect_matching`. CAUTICAL performs better on smaller instances of `register_allocation` and `mutilated_chessboard`, while PRELEARN performs better on larger instances. CAUTICAL can also solve instances of `satcoin` that PRELEARN cannot.

D. Heuristics

Figure 7 shows the performance of CAUTICAL with different heuristics (discussed in Section IV) turned on and off. We evaluate on the different benchmark families discussed in Subsection V-C.

First we consider turning three optimizations off. Figure 7a compares CAUTICAL to CAUTICAL with `shrink` turned off, i.e. we do not shrink PR clauses. Figure 7b compares CAUTICAL to CAUTICAL with `filter-triv` off, i.e. we do not filter trivial clauses. Figure 7c compares CAUTICAL to CAUTICAL with `filter-long` set to 10, i.e. we filter out clauses of size > 10 (as opposed to 2 by default).

When we disable any of three optimizations, the performance is significantly worse, especially on `perfect_matching`, `mutilated-chessboard`, and `register-allocation` benchmarks.

Next, we consider turning three potential “optimizations” on. Figure 7d sets `longer-preprocess` to 100 seconds (as opposed to 30 seconds by default). Figure 7e turns on `order-i`, propagating the first literal i ordered by which literals occurs most frequently in the original formula. Finally, Figure 7 turns

on **select-j**, picking j the second literal by whether it “touches” the first literal i .

Enabling any of these three “optimizations” does not improve performance and in the case of **order-i**, it slightly hurts performance.

E. Research Questions

To conclude, we discuss the two research questions.

For RQ1, CAUTICAL decisively outperforms all solvers except SADICAL on the pigeonhole benchmarks. On SAT competition benchmarks, CAUTICAL improves CADICAL’s PAR-2 score by 2.2%. PRELEARN performs even better, improving CADICAL’s PAR-2 score by 5.4%. However, on formulas where CAUTICAL learns a PR clause, it improves CADICAL’s runtime by 5% or more on 29.9% of benchmarks. On formulas where PRELEARN learns a PR clause, it improves CADICAL’s runtime by 5% or more on 24.7% of benchmarks.

For RQ2, CAUTICAL is the only solver to solve all pigeonhole formulas after scanfilization. Indeed, scanfilization has a negligible effect on CAUTICAL’s performance.

We experienced some success with robustness on the SAT competition benchmarks, but not as significant as for pigeonhole. For instance, in the initial stage when propagating first on literal i (see Algorithm 1), we randomly pick the order for literal i .

Prior PR learning approaches such as PRELEARN used specific orderings of literals to their advantage. We evaluated such an ordering in Figure 7e and found that it did not make a difference for CAUTICAL.

VI. FUTURE WORK

We believe conditional autarkies provide a route to incorporate PR learning as a regular part of state-of-the-art SAT solvers such as CADICAL. However, there are a few important limitations that need to be addressed first.

- 1) **Improving the algorithms:** The heuristics in CAUTICAL are designed to be modular and easy to modify. However, they can be directly incorporated into the main algorithm. For instance, we could embed the binary clause filter by modifying the greedy set cover to discard clauses with more than 2 literals.
- 2) **Shortest proofs for specific families:** Prior work has shown $O(n^3)$ PR proofs of mutilated chessboard [24]. While CAUTICAL learns useful clauses to speed up solving for mutilated chessboard, it is unknown whether conditional autarkies can match the $O(n^3)$ complexity. Similar questions exist for other well-known families such as Tseitin formulas over expander graphs [25, 39].
- 3) **Learning PR clauses with an inprocessing step:** We experimented with learning PR clauses via inprocessing (instead of preprocessing). However, the current PR proof system defines a PR clause as in Definition 1 where Γ is the set of all clauses in the formula, including the learnt, redundant clauses. However, not all redundant clauses

must be considered. This could produce shorter clauses via conditional autarky. However, to realize this, we must modify the PR proof checker.

VII. CONCLUSION

PR clause learning is effective for learning short proofs for difficult problems. However, current PR clause learning techniques require an NP-hard check. We solve this by providing a technique that learns PR clauses in a linear time preprocessing step. We provide clause shrinking and filtering techniques to ensure that we learn useful PR clauses.

Our implementation, CAUTICAL, provides short PR proofs for the pigeonhole principle matching the best possible results. While prior work is only effective on specific pigeonhole encodings, CAUTICAL finds these proofs even when the formula is scanfilized.

Additionally, CAUTICAL is effective on a number of benchmarks from the SAT competition, including most of the families identified in Subsection V-C.

In the future, we hope that PR clause learning can find its way into the main branch of a popular SAT solver. We believe this work is an effective step in this direction.

VIII. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by funding from AFRL and DARPA under Agreement FA8750-24-9-1000. Joseph Reeves was supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

REFERENCES

- [1] N. Rungta, “A billion SMT queries a day,” 2022.
- [2] A. Gupta, M. K. Ganai, and C. Wang, “SAT-based verification methods and applications in hardware verification,” in *Formal Methods for Hardware Verification*, 2006.
- [3] A. R. Bradley, “Understanding IC3,” in *Theory and Applications of Satisfiability Testing (SAT)*, 2012.
- [4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal Methods in System Design*, vol. 19, pp. 7–34, 2001.
- [5] J. Rintanen, “Heuristics for planning with SAT,” in *Principles and Practice of Constraint Programming (CP)*, D. Cohen, Ed., 2010.
- [6] H. Kautz and B. Selman, “Planning as satisfiability,” in *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, 1992.
- [7] M. Soos, K. Nohl, and C. Castelluccia, “Extending SAT solvers to cryptographic problems,” in *Theory and Applications of Satisfiability Testing (SAT)*, 2009.
- [8] 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 (TACAS), 2023.

- [9] M. J. H. Heule, O. Kullmann, and V. W. Marek, “Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer,” in *Theory and Applications of Satisfiability Testing (SAT)*, 2016.
- [10] J. Brakensiek, M. Heule, J. Mackey, and D. Narváez, “The resolution of Keller’s conjecture,” *Journal of Automated Reasoning (JAR)*, vol. 66, no. 3, pp. 277–300.
- [11] M. J. H. Heule and M. Scheucher, “Happy ending: An empty hexagon in every set of 30 points,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2024.
- [12] M. J. H. Heule, B. Kiesl, and A. Biere, “Short proofs without new variables,” in *Conference on Automated Deduction (CADE)*, 2017.
- [13] A. Haken, “The intractability of resolution,” *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985.
- [14] M. J. H. Heule, B. Kiesl, and A. Biere, “Encoding redundancy for satisfaction-driven clause learning,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019.
- [15] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, “Cadical 2.0,” in *Computer Aided Verification (CAV)*, 2024.
- [16] A. Fleury and M. Heisinger, “Cadical, kissat, paracooba, plingeling and treengeling entering the SAT competition 2020,” *SAT Competition*, 2020.
- [17] A. Biere, “Lingeling, plingeling, picosat and precosat at SAT race 2010,” 2010.
- [18] B. Kiesl, M. J. H. Heule, and A. Biere, “Truth assignments as conditional autarkies,” in *Automated Technology for Verification and Analysis (ATVA)*, 2019.
- [19] O. Kullmann, “On a generalization of extended resolution,” *Discrete Applied Mathematics*, vol. 96–97, pp. 149–176, 1999.
- [20] M. Järvisalo, M. J. H. Heule, and A. Biere, “Inprocessing rules,” in *International Joint Conference on Automated Reasoning (IJCAR)*, 2012.
- [21] B. Kiesl, M. Seidl, H. Tompits, and A. Biere, “Super-blocked clauses,” in *International Joint Conference on Automated Reasoning (IJCAR)*, 2016.
- [22] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, “Pruning through satisfaction,” in *Hardware and Software: Verification and Testing (HVC)*, 2017.
- [23] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [24] M. J. H. Heule, B. Kiesl, and A. Biere, “Clausal proofs of mutilated chessboards,” in *NASA Formal Methods (NFM)*, 2019.
- [25] A. Urquhart, “Hard examples for resolution,” *Journal of the Association for Computing Machinery (JACM)*, vol. 34, no. 1, p. 209–219, 1987.
- [26] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, “Preprocessing of Propagation Redundant Clauses,” *Journal of Automated Reasoning (JAR)*, vol. 67, no. 3, p. 31, 2023.
- [27] P. Slavík, “A tight analysis of the greedy algorithm for set cover,” in *Symposium on Theory of Computing (STOC)*, 1996.
- [28] T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023.
- [29] M. J. H. Heule, “dpr-trim: Checker of dpr proof files,” <https://github.com/marijnheule/dpr-trim>, accessed: May 1, 2025.
- [30] X. C. Song, P. Smith, R. Kalyanam, X. Zhu, E. Adams, K. Colby, P. Finnegan, E. Gough, E. Hillery, R. Irvine, A. Maji, and J. St. John, “Anvil - system architecture and experiences from deployment and early user operations,” in *Practice and Experience in Advanced Research Computing (PEARC)*, 2022.
- [31] T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, 2022.
- [32] M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, 2024.
- [33] A. Biere and M. Heule, “The effect of scrambling cnfs,” in *Pragmatics of SAT (PoS)*, 2019.
- [34] M. Alekhovich, “Mutilated chessboard problem is exponentially hard for resolution,” *Theoretical Computer Science*, vol. 310, no. 1, pp. 513–525, 2004.
- [35] C. R. Codel, J. Reeves, M. J. H. Heule, and R. E. Bryant, “Bipartite perfect matching benchmarks,” in *Pragmatics of SAT (PoS)*, 2021.
- [36] A. Haberlandt and H. Green, “Python function register allocation benchmarks,” in *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023, p. 56.
- [37] J. Chung, S. Buss, and V. Ganesh, “Unsatcoin,” in *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023, p. 52.
- [38] A. Biere, “CNF encodings of complete pairwise combinatorial testing of our SAT solver SATCHe,” in *Proceedings of SAT Competition 2021: Solver, Benchmark and Proof Checker Descriptions*, 2023, p. 46.
- [39] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pp. 466–483, 1983.

R2U2 Playground: Visualization of a Real-time, Temporal Logic Runtime Monitor

Alexis Aurandt 
Iowa State University
Ames, IA, USA
aurandt@iastate.edu

Kristin Yvonne Rozier 
Iowa State University
Ames, IA, USA
kyrozier@iastate.edu

Phillip H. Jones 
Iowa State University
Ames, IA, USA
phjones@iastate.edu



Abstract—The Realizable, Responsive, Unobtrusive Unit (R2U2) is a real-time, temporal logic-based runtime monitoring engine that has been successfully deployed on-board a wide range of cyber-physical systems, from aircraft to spacecraft to robots, checking in real time whether these systems uphold specified system requirements. However, the efficacy of deploying runtime monitors is highly sensitive to their correct configuration. Moreover, there are many barriers to adopting runtime monitors, including a high learning curve and the challenge of eliciting formal specifications that accurately capture system requirements. Therefore, we present the R2U2 Playground, an interactive web-based playground that provides visualization of R2U2. The R2U2 Playground provides stepwise execution coupled with reactive timeline plotting and visualization of its internal abstract syntax tree architecture. To this extent, the R2U2 Playground provides insight into how R2U2 evaluates specifications, allowing for easier specification understanding and debugging.

I. INTRODUCTION

The Realizable, Responsive, Unobtrusive Unit (R2U2) runtime monitoring engine is known for its real-time guarantees and resource-aware architecture [1]–[5]. Hence, R2U2 has been successfully deployed on-board various cyber-physical systems such as NASA’s Robonaut2 [1], NASA’s Lunar Gateway Vehicle System Manager [6]–[9], NASA’s Swift UAS [2], [10], [11], NASA’s DragonEye UAS [12], Iowa State’s Nova Somnium sounding rocket [13], Iowa State’s CySat-I CubeSat [14], and more [15]–[19]. R2U2 successfully analyzed whether these systems upheld formal system requirements during deployment. However, R2U2’s efficacy is highly sensitive to its correct configuration and integration into the target system.

Furthermore, there are many barriers to adopting runtime monitors, such as the learning barrier, skepticism, lack of experience, etc. [20]–[22]. Additionally, once runtime monitors are adopted, there is the challenge of eliciting specifications that accurately capture system requirements [23]–[25]. To decrease the learning curve of the R2U2 runtime monitoring engine, we present the R2U2 Playground.¹² The R2U2 Playground is an interactive web-based playground that provides a visualization of the R2U2 runtime monitoring engine. Within the playground, an instantiation of R2U2 is executed given

user-defined specifications and an input trace. The R2U2 Playground eases the elicitation and testing of these user-defined specifications by providing step-forward and step-backward debugging capabilities and timeline visualization of output verdicts produced by R2U2. It also provides a visualization for how R2U2 reasons about a (set of) specification(s) as an abstract syntax tree, and we enable configuring of various optimizations of the abstract syntax tree (e.g., common sub-expression elimination [1], [5] and rewrite rules [26]) directly in the playground. The playground interface also includes easy import and export of the various input/output files, visualization images, and the compiled specification binary. Overall, the R2U2 Playground allows users to easily interact with R2U2, learn how R2U2 reasons about specification(s), and, as a result, also validate the correctness of their specification(s) and R2U2’s configuration.

II. CURRENT VISUALIZATIONS FOR RUNTIME MONITORS

Within the runtime verification community, system requirements are specified/formalized utilizing two main approaches: (1) stream-based specification languages and (2) temporal logics. The utilization of a stream-based specification language to specify system requirements to generate runtime monitors is seen in frameworks such as RTLola [27], [28], Striver [29], TeSSLa [30], and CoPilot [31], [32]. These stream-based specification languages are arguably more expressive than temporal logics, but they have a high learning curve as they are each tool-specific languages. To combat the learning curve, both RTLola [33] and TeSSLa [30] have interactive playgrounds for exploring their respective tools and to ease the understanding of specifications. On the other hand, other tools look at monitoring temporal logics directly, e.g., R2U2 [5], [34], Aerial [35], Reelay [36], MonPoly [37], [38], VeriMon [39], VeriMon+ [40], Hydra [41], and Vydra [42], but these monitors currently lack any explainability or visualization interface/tool. Recently, the Explanator2 [43] and WhyMon [44] monitors were developed to bridge this gap in explainability through outputting proof trees and providing optional GUI visualization of these proof trees for Metric Temporal Logic (MTL) and Metric First-Order Temporal Logic (MFOTL) specifications, respectively. Of these temporal logic monitors, R2U2 is the only one that can run on-board and provide

Supported by NSF:CPS Award 2038903.

¹<https://r2u2.temporallogic.org/playground/>

²<https://zenodo.org/records/16787011>

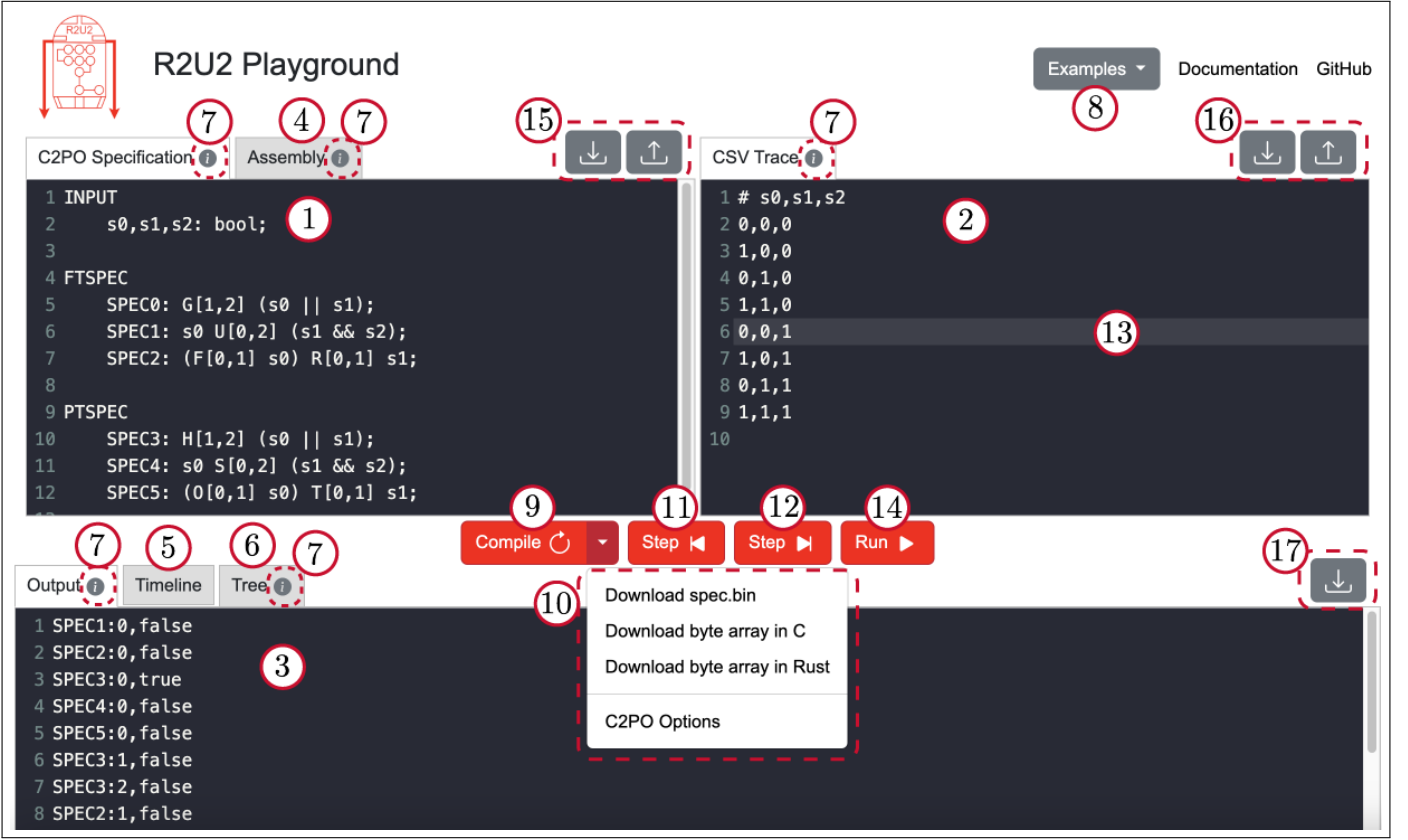


Fig. 1: R2U2 Playground: (1) C2PO specification; (2) CSV input trace; (3) R2U2 text output; (4) C2PO assembly output tab; (5) R2U2 timeline visualization tab; (6) R2U2 internal tree architecture visualization tab; (7) Info pop-up buttons; (8) Examples dropdown; (9) Compile C2PO specification button; (10) Compile dropdown menu; (11) Step-backward button; (12) Step-forward button; (13) Trace row highlighting based on current step; (14) Run button; (15) Download/Upload C2PO specification buttons; (16) Download/Upload CSV trace buttons; (17) Download output text, timeline image, or tree image button.

verdicts in real-time for both future- and past-time properties due to its unique architecture and specification language [6]; therefore, we focus on providing a visualization for R2U2, while also taking inspiration from the TeSSLa and RTLola playgrounds.

III. PLAYGROUND FUNCTIONALITY

A. Mission-time Linear Temporal Logic (MLTL) and past-time MLTL (ptMLTL) [2], [34], [45], [46]

R2U2 reasons over specifications in MLTL/ptMLTL, a variant of LTL/ptLTL over finite traces with bounded, closed, and discrete intervals on temporal operators. MLTL and ptMLTL contain the standard boolean connectives (e.g., NOT, AND, OR, XOR, IMPLIES, EQUIVALENT) along with temporal operators that reason over future-time and past-time, respectively. Tables I and II provide a visualization of example formulas containing these temporal operators.

B. Configuration Compiler for Property Organization (C2PO) [5] Specification and R2U2 Configuration Output

C2PO provides a specification language to make writing MLTL/ptMLTL specifications more natural and transparent.

Notably, C2PO supports expressions over complex data types such as sets and structs. The language is also structured to include separate sections for defining structs, inputs, variables, macros, and MLTL/ptMLTL formulas. Within the R2U2 Playground, users can directly compose their C2PO specification (label 1 in Fig. 1), as well as upload/download C2PO specification files directly (label 15 in Fig. 1).

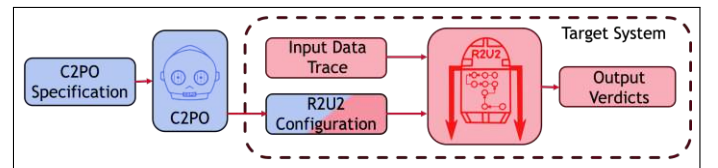


Fig. 2: C2PO and R2U2 Workflow: Blue shaded boxes indicate the process required to configure R2U2, and the red shaded boxes indicate the pieces required to execute R2U2.

C2PO encodes a C2PO specification into an R2U2 configuration by decomposing the specified expression(s) into sub-expression(s) represented as an Abstract Syntax Tree (AST) and traverses the AST to produce assembly-style instructions for R2U2 to execute at runtime (Fig. 2). (Refer to Fig. 3 and 4 for

TABLE I: Pictorial Representation of Mission-time Linear Temporal Logic (MLTL). Timeline depicts a satisfying trace where $i = 0$.

Formula	Operator	Timeline
$\mathcal{G}_{[2,5]} \psi$	globally _[2,5]	
$\mathcal{F}_{[0,6]} \psi$	future _[0,6]	
$\psi \mathcal{U}_{[1,4]} \xi$	until _[1,4]	
$\psi \mathcal{R}_{[3,6]} \xi$	release _[3,6]	

TABLE II: Pictorial Representation of past-time Mission-time Linear Temporal Logic (ptMLTL). Timeline depicts a satisfying trace where $i = 6$.

Formula	Operator	Timeline
$\mathcal{H}_{[2,5]} \psi$	historically _[2,5]	
$\mathcal{O}_{[0,6]} \psi$	once _[0,6]	
$\psi \mathcal{S}_{[1,4]} \xi$	since _[1,4]	
$\psi \mathcal{T}_{[3,6]} \xi$	trigger _[3,6]	

the AST and assembly-style instructions required to encode SPEC0 from the C2PO specification in Fig. 1.) The R2U2 configuration (which is composed of these assembly-style instructions) is represented in binary format. The command line interfaces for both R2U2's C and Rust realizations expect the R2U2 configuration as a binary file, while if running R2U2 on-board a system, it may be more applicable to have the binary stored as an array; therefore, a user can easily download the R2U2 configuration as a binary file (i.e., spec.bin) or as a C/Rust file containing the configuration encoded as a byte array (label 10 in Fig. 1).

C2PO Specification	Assembly
1 BZ b0	iload 0 0
2 BZ b1	store 0 0
3 BZ b2	iload 1 0
4 BZ b3	store 2 1
5 TL n0	load a0
6 TL n1	load a1
7 TL n2	or n0 n1
8 TL n3	release False n2
9 TL n4	return n3 0
10 CG TL SCQ q0	1
11 CG TL SCQ q1	1
12 CG TL SCQ q2	1

Fig. 3: Textual Representation of Assembly Output for SPEC0: $\mathcal{G}[1,2] (s0 \parallel s1)$

Once the C2PO file has been compiled in the playground by either pressing the Compile button (label 9 in Fig. 1) or by executing R2U2 (labels 11, 12, and 14 in Fig. 1), the textual representation of these instructions is displayed in the

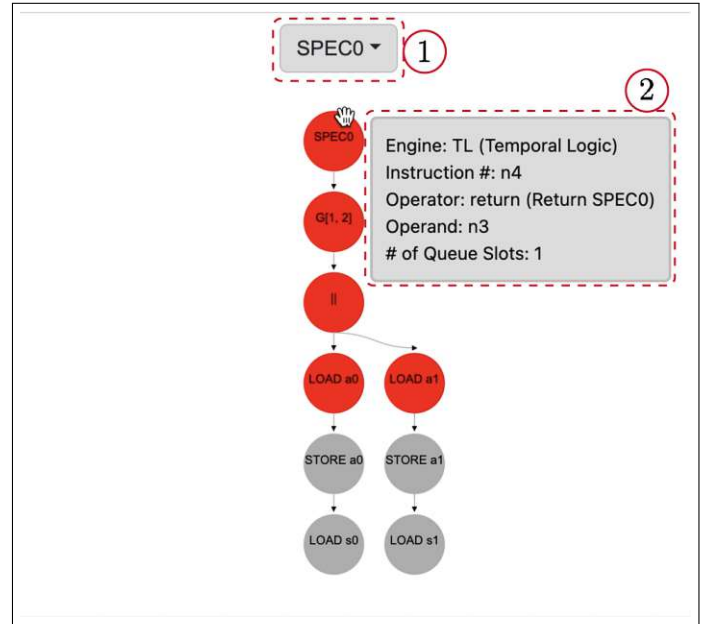


Fig. 4: Tree Visualization of SPEC0: $\mathcal{G}[1,2] (s0 \parallel s1)$: (1) Specification selection dropdown menu; (2) More information dialog box displayed upon mouse over of node.

Assembly tab (label 4 in Fig. 1). Fig. 3 displays the Assembly tab if only SPEC0 was defined in the C2PO specification given in Fig. 1. This textual representation may be difficult to parse; therefore, a visual representation of the AST is available in the Tree tab (label 6 in Fig. 1). Fig. 4 displays the Tree tab with SPEC0 selected, where SPEC0 was defined in the C2PO specification given in Fig. 1. Within the Tree tab, a dropdown menu (label 1 in Fig. 4) allows a user to select any specification that has been defined, or they can select to view all specifications at once (as shown in Fig. 6). The tree is also color-coded such that red nodes represent MLTL/ptMLTL operator nodes, while grey nodes represent Booleanizer [5] nodes. (The Booleanizer is an optional module that computes atomic propositions from non-Boolean expressions utilizing arithmetic, bitwise, set aggregation, and relational operators.) A user can also zoom in and out of the AST view, move the entire tree around, move individual nodes around, and mouse over nodes to find out more information (label 2 in Fig. 4). Optionally, a user may also download the image to save locally (label 17 in Fig. 1).

C2PO also allows users to enable or disable certain features (i.e., the Booleanizer, auxiliary data, various rewriting rules [26], Common Subexpression Elimination (CSE) [1], [5], and SAT checking) to customize the AST for a given target system. A user can directly enable/disable these features by clicking on the C2PO Options button from the Compile dropdown (label 10 in Fig. 1) to bring up the C2PO options pop-up as shown in Fig. 5. Fig. 6 illustrates how the AST may change based on the features enabled. Fig. 6a displays the AST for all specifications defined in the C2PO specification from Fig. 1 based on the default C2PO options displayed in Fig. 5, and Fig. 6b displays

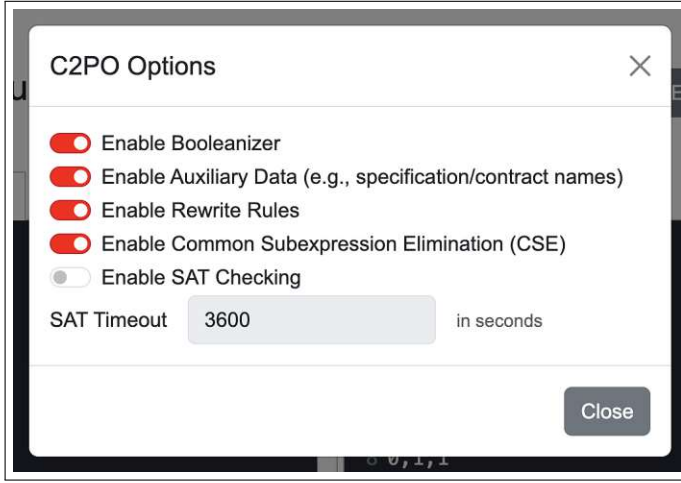
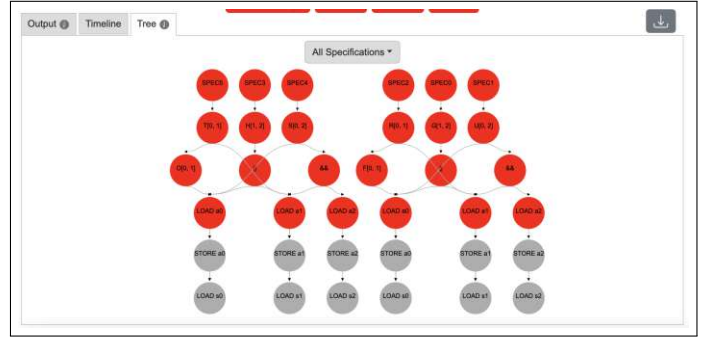


Fig. 5: C2PO Options Pop-Up Window: (a) Enable Booleanizer [5] module; (b) Enable Auxiliary Data such as specification names (e.g., “SPEC0”) to be stored and provided with output verdicts; (c) Enable Rewrite Rules given in [26]; (d) Enable CSE allows to share common nodes between subexpressions [1], [5]; (e) Enable SAT Checking of the MLTL/ptMLTL specifications powered by Z3 [47].

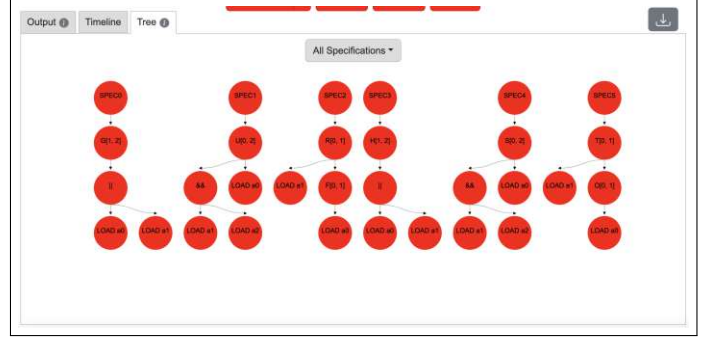
the resulting AST when the Booleanizer and CSE are disabled. The specifications given in the C2PO specification in Fig. 1 only reason over Boolean inputs; therefore, the Booleanizer is not required for this C2PO specification and can be disabled to save resources as shown by the elimination of the grey nodes in Fig. 6b. On the other hand, if we disable CSE, we no longer share common nodes between subexpressions; therefore, we also observe a larger amount of red nodes in Fig. 6b.

C. R2U2 Execution

There are three options for executing the R2U2 runtime monitoring engine: step-forward (label 11 in Fig. 1), step-backward (label 12 in Fig. 1), and run (label 14 in Fig. 1). To define the inputs into R2U2, a CSV formatted input signal trace can be composed directly within the playground (label 2 in Fig. 1), as well as uploaded/downloaded (label 16 in Fig. 1). Within the CSV trace, each line typically represents the input signals at a discrete timestamp, but to support longer traces as requested by users at NASA (without having to manually specify the input of each time-step), we added the ability to encode multiple time-steps in a single line where the symbol $@T$ at the beginning of a line encodes the input signal as occurring at time T and since the last time-step included in the input trace. This format directly works with R2U2’s aggregated storing of results (refer to [1] and [34] for more details on R2U2’s aggregated storage). The $@T$ symbol is optional, and if not included, the line simply represents the next timestamp of execution. Fig. 7 provides an example of employing the $@T$ symbol. Utilizing the step-forward and step-backward buttons, the CSV trace is iterated through step-by-step, and the current step in the CSV trace is highlighted



(a) Booleanizer and Common Subexpression Elimination (CSE) are enabled



(b) Booleanizer and CSE are disabled

Fig. 6: Tree Visualization of All Specifications (i.e., SPEC0, SPEC1, SPEC2, SPEC3, SPEC4, and SPEC5 from Fig. 1)

```
# p,q,i,j,k
0,1,5,2,3
0,1,5,2,3
0,1,5,2,3
0,0,4,2,3
0,0,4,2,3
1,1,2,6,10
```

(a) Typical trace where each line represents a sequential discrete timestamp starting at timestamp 0

```
# p,q,i,j,k
@2 0,1,5,2,3
@4 0,0,4,2,3
@5 1,1,2,6,10
```

(b) Trace encoding multiple time-steps utilizing the $@T$ symbol

Fig. 7: Different Equivalent CSV Input Trace Representations: Note that “# p, q, i, j, k ” defines the variable-to-column mapping scheme for the trace.

(label 13 in Fig. 1). Additionally, the entire CSV trace can be evaluated at once using the run button.

The textual representation of R2U2’s output verdicts is given in the Output tab (label 3 in Fig. 1), but a timeline visualization is provided in the Timeline tab (label 5 in Fig. 1). Fig. 8 displays the Timeline tab with SPEC0 selected, but a dropdown menu (label 1 in Fig. 8) allows a user to visualize

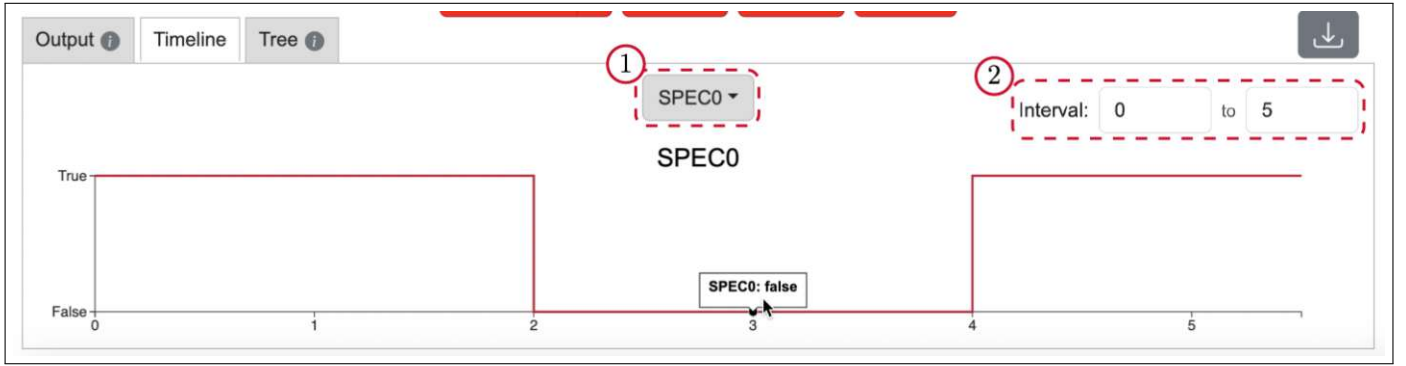


Fig. 8: Timeline Visualization of SPEC0: $G[1,2] (s_0 \parallel s_1)$: (1) Specification selection dropdown menu; (2) Interval zooming.

any specification that has been defined. When performing stepwise execution, both the textual representation and the timeline visualization will react accordingly, allowing the user to visualize how R2U2 produces output verdicts over time. Optionally, the user can also zoom in to a certain interval of interest (label 2 in Fig. 8) and download the timeline image to save locally (label 17 in Fig. 1).

IV. IMPLEMENTATION

The R2U2 Playground is comprised of two main components: the frontend and the backend. The frontend consists of the visual user interface as shown in Fig. 1. The frontend is strictly written in HTML, JavaScript, and CSS with the aid of Bootstrap [48], CodeMirror [49], and D3 [50]. The frontend purely provides the visualization elements of the R2U2 Playground, while the execution of the R2U2 runtime monitoring engine occurs on the backend. The backend is a Rust-based server built with the warp framework [51] that directly executes the Rust realization of R2U2 [34]. (Note that we could have easily utilized the R2U2’s C realization in the backend but chose the Rust realization due to its memory-safe qualities and ease-of-use.) The frontend sends applicable requests with the C2PO specification and CSV input trace to the backend, and the backend will then respond with the applicable response (e.g., R2U2 configuration and verdicts). The frontend then performs static analysis on the response to provide its various visualizations.

To promote extensibility, the R2U2 Playground is also available open-source under the Creative Commons Attribution 4.0 International license.³

V. DISCUSSION

The R2U2 Playground allows users to easily run C2PO specifications against various CSV input traces without requiring local installation or interacting with command line interfaces. The playground is also easy to use with several optional examples available through the Example dropdown (label 8 in Fig. 1) and optional more-information pop-up windows (label 7 in Fig. 1). Consequently, the R2U2 Playground lowers the barrier to entry for utilizing the R2U2

runtime monitoring engine, allowing for easier adoption of R2U2. To this extent, we have witnessed the easier adoption of R2U2 through sharing the R2U2 Playground with various interested industrial entities (e.g., Collins Aerospace, Kansas State University, and NASA).

Through the playground, users can more easily debug their specifications through stepwise debugging capabilities and reactive timeline visualization. Additionally, users can enable or disable C2PO features and view how this changes the internal representation (i.e., the AST) of their specifications in R2U2. Previously, the Resource Estimation GUI from [5] provided limited capabilities to perform similar analysis, but the information was not intuitively displayed. Additionally, this previous work only calculated resource usage for running R2U2 and did not provide an interface to execute R2U2 or debug specifications. In the future, we plan to incorporate the resource estimation analysis of this previous work into the R2U2 Playground to visualize the resource requirements of the AST. Furthermore, if a user’s end goal is to run R2U2 on-board a system outside the R2U2 Playground, they can easily export their compiled specifications to a range of popular formats R2U2 might expect (e.g., a binary file or a byte array in C or Rust). Overall, the R2U2 Playground can be utilized to both learn how R2U2 reasons about specifications and debug/test these specifications to make sure they are accurately capturing the system requirements, while also ensuring the correct configuration of R2U2.

Given the prevalence of MLTL/ptMLTL as a specification logic and the popularity of runtime monitoring as a lightweight formal verification technology, the impact of the R2U2 Playground also extends beyond R2U2, providing a tool for learning MLTL/ptMLTL and runtime monitoring. We utilized the R2U2 Playground as an educational tool for teaching these aspects of formal methods in the 2025 Summer School on Formal Techniques [52], and in the Applied Formal Methods course at Iowa State University [53]. Future expansions of the R2U2 Playground will include additional educational tooling in response to questions and feedback from students.

REFERENCES

- [1] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier, “Embedding Online Runtime Verification for Fault Disambiguation on

³<https://zenodo.org/records/16787011>

- Robonaut2,” in *Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, ser. Lecture Notes in Computer Science (LNCS). Vienna, Austria: Springer, September 2020, pp. 196–214.
- [2] T. Reinbacher, K. Y. Rozier, and J. Schumann, “Temporal-logic based runtime observer pairs for system health management of real-time systems,” in *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science (LNCS), vol. 8413. Springer-Verlag, April 2014, pp. 357–372.
 - [3] J. Schumann, P. Moosbrugger, and K. Y. Rozier, “Runtime Analysis with R2U2: A Tool Exhibition Report,” in *Proceedings of the 16th International Conference on Runtime Verification (RV16)*. Madrid, Spain: Springer-Verlag, September 2016.
 - [4] K. Y. Rozier and J. Schumann, “R2U2: Tool Overview,” in *Proceedings of International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES)*, vol. 3. Seattle, WA, USA: Kalpa Publications, September 2017, pp. 138–156. [Online]. Available: <https://easychair.org/publications/paper/Vncw>
 - [5] C. Johannsen, P. Jones, B. Kempa, K. Y. Rozier, and P. Zhang, “R2U2 Version 3.0: Re-Imagining a Toolchain for Specification, Resource Estimation, and Optimized Observer Generation for Runtime Verification in Hardware and Software,” in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 483–497.
 - [6] J. B. Dabney, J. M. Badger, and P. Rajagopal, “Adding a verification view for an autonomous real-time system architecture,” in *Proceedings of SciTech Forum*, ser. 2021-0566. AIAA, January 2021, p. Online.
 - [7] J. B. Dabney, “Using assume-guarantee contracts in autonomous spacecraft,” Flight Software Workshop (FSW) Online: <https://www.youtube.com/watch?v=zrtyiyNf674>, February 2021.
 - [8] J. B. Dabney, P. Rajagopal, and J. M. Badger, “Using assume-guarantee contracts for developmental verification of autonomous spacecraft,” Flight Software Workshop (FSW) Online: <https://www.youtube.com/watch?v=HFnn6TzblPg>, February 2022.
 - [9] J. B. Dabney, J. M. Badger, and P. Rajagopal, “Trustworthy autonomy for gateway vehicle system manager,” in *2023 IEEE Space Computing Conference (SCC)*. IEEE, 2023, pp. 57–62.
 - [10] J. Geist, K. Y. Rozier, and J. Schumann, “Runtime observer pairs and bayesian network reasoners on-board FPGAs: flight-certifiable system health management for embedded systems,” in *International Conference on Runtime Verification*. Springer, 2014, pp. 215–230.
 - [11] J. Schumann, K. Y. Rozier, T. Reinbacher, O. J. Mengshoel, T. Mbaya, and C. Ippolito, “Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems,” in *International Journal of Prognostics and Health Management*, 2015.
 - [12] J. Schumann, P. Moosbrugger, and K. Y. Rozier, “R2U2: monitoring and diagnosis of security threats for unmanned aerial systems,” in *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*. Springer, 2015, pp. 233–249.
 - [13] B. Hertz, Z. Luppen, and K. Y. Rozier, “Integrating runtime verification into a sounding rocket control system,” in *NASA Formal Methods Symposium*. Springer, 2021, pp. 151–159.
 - [14] A. Aurandt, P. H. Jones, and K. Y. Rozier, “Runtime verification triggers real-time, autonomous fault recovery on the CySat-I,” in *NASA Formal Methods Symposium*. Springer, 2022, pp. 816–825.
 - [15] M. Cauwels, A. Hammer, B. Hertz, P. Jones, and K. Y. Rozier, “Integrating Runtime Verification into an Automated UAS Traffic Management System,” in *DETECT*. L’Aquila, Italy: Springer, September 2020.
 - [16] C. Johannsen, M. Anderson, W. Burken, E. Diersen, J. Edgren, C. Glick, S. Jou, A. Kumar, J. Levandowski, E. Moyer *et al.*, “Openuas version 1.0,” in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2021, pp. 1449–1458.
 - [17] Z. A. Luppen, D. Y. Lee, and K. Y. Rozier, “A case study in formal specification and runtime verification of a cubesat communications system,” in *AIAA Scitech 2021 forum*, 2021.
 - [18] Z. Luppen, M. Jacks, N. Baughman, M. Stilic, R. Nasers, B. Hertz, J. Cutler, D.-Y. Lee, and K. Y. Rozier, “Elucidation and analysis of specification patterns in aerospace system telemetry,” in *NASA Formal Methods Symposium*. Springer, 2022, pp. 527–537.
 - [19] J. Schumann, I. Roychoudhury, and C. Kulkarni, “Diagnostic reasoning using prognostic information for unmanned aerial systems,” in *Annual Conference of the PHM Society*, vol. 7, no. 1, 2015.
 - [20] J. P. Bowen and M. G. Hinchey, “Ten commandments of formal methods,” *Computer*, vol. 28, no. 4, pp. 56–63, 1995.
 - [21] —, “Ten commandments of formal methods... ten years later,” *Computer*, vol. 39, no. 1, pp. 40–48, 2006.
 - [22] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM computing surveys (CSUR)*, vol. 41, no. 4, pp. 1–36, 2009.
 - [23] A. Goodloe, “Challenges in high-assurance runtime verification,” in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 446–460.
 - [24] K. Y. Rozier, “Specification: The biggest bottleneck in formal methods and autonomy,” in *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*. Springer, 2016, pp. 8–26.
 - [25] L. Teixeira, B. Miranda, H. Rebêlo, and M. d’Amorim, “Demystifying the challenges of formally specifying API properties for runtime verification,” in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 82–93.
 - [26] C. Johannsen, B. Kempa, P. H. Jones, K. Y. Rozier, and T. Wongpiromsarn, “Impossible made possible: Encoding intractable specifications via implied domain constraints,” in *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2023, pp. 151–169.
 - [27] J. Baumeister, B. Finkbeiner, F. Kohn, and F. Scheerer, “A tutorial on stream-based monitoring,” in *International Symposium on Formal Methods*. Springer, 2024, pp. 624–648.
 - [28] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah, “StreamLAB: stream-based monitoring of cyber-physical systems,” in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 421–431.
 - [29] F. Gorostiaga and C. Sánchez, “Striver: Stream runtime verification for real-time event-streams,” in *International Conference on Runtime Verification*. Springer, 2018, pp. 282–298.
 - [30] H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, and A. Weiss, “TeSSLa—an ecosystem for runtime verification,” in *International Conference on Runtime Verification*. Springer, 2022, pp. 314–324.
 - [31] I. Perez, A. E. Goodloe, and F. Dedden, “Runtime verification in real-time with the Copilot language: A tutorial,” in *International Symposium on Formal Methods*. Springer, 2024, pp. 469–491.
 - [32] L. Pike, A. Goodloe, R. Morisset, and S. Niller, “Copilot: A hard real-time runtime monitor,” in *International Conference on Runtime Verification*. Springer, 2010, pp. 345–359.
 - [33] B. Finkbeiner, F. Kohn, and M. Schledjewski, “Leveraging static analysis: An IDE for RTLola,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2023, pp. 251–262.
 - [34] A. Aurandt, P. H. Jones, and K. Y. Rozier, “Towards a Safe, Verified Runtime Monitor for Embedded Systems: R2U2 in Embedded Rust,” in *NASA Formal Methods Symposium*. Springer, 2025, pp. 31–53.
 - [35] D. A. Basin, S. Krstić, and D. Traytel, “AERIAL: Almost event-rate independent algorithms for monitoring metric regular properties,” *RV-CuBES*, vol. 3, pp. 29–36, 2017.
 - [36] D. Ulus, “Online monitoring of metric temporal logic using sequential networks,” *arXiv preprint arXiv:1901.00175*, 2019.
 - [37] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu, “MONPOLY: Monitoring usage-control policies,” in *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2*. Springer, 2012, pp. 360–364.
 - [38] D. A. Basin, F. Klaedtke, and E. Zălinescu, “The MonPoly monitoring tool,” *RV-CuBES*, vol. 3, pp. 19–28, 2017.
 - [39] J. Schneider, D. Basin, S. Krstić, and D. Traytel, “A formally verified monitor for metric first-order temporal logic,” in *Runtime Verification: 19th International Conference, RV 2019, Porto, Portugal, October 8–11, 2019, Proceedings 19*. Springer, 2019, pp. 310–328.
 - [40] D. Basin, T. Dardinier, L. Heimes, S. Krstić, M. Raszky, J. Schneider, and D. Traytel, “A formally verified, optimized monitor for metric first-order dynamic logic,” in *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part I 10*. Springer, 2020, pp. 432–453.
 - [41] M. Raszky, D. Basin, S. Krstić, and D. Traytel, “Multi-head monitoring of metric temporal logic,” in *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 151–170.

- [42] M. Raszyk, D. Basin, and D. Traytel, “Multi-head monitoring of metric dynamic logic,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 233–250.
- [43] L. Lima, A. Herasimau, M. Raszyk, D. Traytel, and S. Yuan, “Explainable online monitoring of metric temporal logic,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2023, pp. 473–491.
- [44] L. Lima, J. J. Huerta y Munive, and D. Traytel, “WhyMon: A runtime monitoring tool with explanations as verdicts,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2024, pp. 76–90.
- [45] J. Elwing, L. Gamboa-Guzman, J. Sorkin, C. Traveset, Z. Wang, and K. Y. Rozier, “Mission-time LTL (MLTL) formula validation via regular expressions,” in *International Conference on Integrated Formal Methods*. Springer, 2023, pp. 279–301.
- [46] J. Li, M. Y. Vardi, and K. Y. Rozier, “Satisfiability checking for mission-time LTL,” in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 3–22.
- [47] 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*. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
- [48] Mark Otto and Jacob Thornton, “Bootstrap,” 2011–2025. [Online]. Available: <https://getbootstrap.com/docs/5.3/getting-started/introduction/>
- [49] Marijn Haverbeke, “Codemirror,” 2007–2025. [Online]. Available: <http://codemirror.net/>
- [50] Mike Bostock and Observable, Inc., “D3 (or D3.js),” 2011–2025. [Online]. Available: <https://d3js.org/>
- [51] Sean McArthur, “warp,” 2018–2024. [Online]. Available: <https://crates.io/crates/warp>
- [52] SRI, “Summer School on Formal Techniques,” Menlo Park, California, USA, , May 2025.
- [53] K. Y. Rozier, “On teaching applied formal methods in aerospace engineering,” in *Proceedings of the Formal Methods Teaching Workshop (FMTea) at the 3rd World Congress on Formal Methods*, ser. Lecture Notes in Computer Science (LNCS), vol. 11758. Porto, Portugal: Springer, October 2019, pp. 111–131. [Online]. Available: <https://fmtea.github.io/>

s2s: An Eager SMT Solver for Strings

Kevin Lotz 

Department of Computer Science
Kiel University
Kiel, Germany
kel@informatik.uni-kiel.de

Mitja Kulczynski 

Department of Computer Science
Kiel University
Kiel, Germany
mku@informatik.uni-kiel.de

Dirk Nowotka 

Department of Computer Science
Kiel University
Kiel, Germany
dn@informatik.uni-kiel.de

Abstract—String constraint solving describes the problem of determining the satisfiability of first-order formulas where variables range over strings. Automated procedures for solving these problems are known as string solvers. Most existing solvers adopt a lazy SMT approach, where a SAT solver handles the Boolean structure of the formula and alternates with a specialized string reasoning engine, following the CDCL(T) paradigm. An alternative strategy, called eager SMT solving, reduces the entire problem to Boolean satisfiability, allowing it to be handled directly by a SAT solver. While successful eager approaches have been proposed, current implementations either lack expressiveness or are not publicly available. Here, we present a new eager string solver based on existing techniques, capable of solving Boolean combinations of word equations, regular constraints, and linear arithmetic over string lengths. An evaluation on the SMT-LIB string benchmarks shows that our approach is competitive on a broad set of problems compared to state-of-the-art solvers, and even outperforms them in many cases. In particular, our solver demonstrates close to best-in-class performance on the SMT-COMP pure string benchmarks.

I. INTRODUCTION

String constraint solving is the task of determining the satisfiability of first-order formulas over string variables, typically involving constructs such as word equations, regular expression membership, and reasoning about string lengths. String constraints are nearly unavoidable in verification tasks and they often arise in security-sensitive domains such as web application analysis and access policy validation [1], [2].

While the satisfiability of word equations and their combination with regular constraints is known to be decidable [3], [4], the known theoretical decision procedures are far from efficient (e.g., [3]–[6]). Moreover, many combinations of string constraints are undecidable (e.g., word equations with string-number conversion predicates [6]) or of unknown decidability status (e.g., word equations with length constraints). Despite these challenges, numerous practical string solvers have been developed. Most solvers employ the *lazy* SMT paradigm, typically by adapting a CDCL(T) approach, which combines a SAT solver for the propositional structure with a dedicated theory solver for string reasoning [7]–[9].

A contrasting approach is the *eager* SMT methodology, in which all constraints are eagerly reduced to propositional logic and handed off to a SAT solver. Two eager solvers for the theory of strings have been proposed: WOORPJE [10] and NFA2SAT [11]. Building on both, we present s2s, a new eager string solver that combines NFA2SAT’s incremental reduction

approach with WOORPJE’s encoding of integer arithmetic. As a result, s2s supports solving Boolean combinations of word equations, regular expression constraints, and linear arithmetic over string lengths and integer variables.

s2s features a two-stage architecture, beginning with a thorough preprocessing phase that applies various simplifications to cut down the search space as much as possible. For the reduction to propositional logic, s2s relies on a bounded approach. It imposes initially small bounds on all variables and gradually increases them when necessary. To make this efficient, the solver adapts the *incremental encodings* and *unsatisfiable core analysis* introduced by NFA2SAT [11]. To handle arithmetic constraints, s2s adopts the approach of WOORPJE, compiling them into propositional logic using multivariate decision diagrams, which compactly represent the sets of bounded integer solutions.

We evaluated s2s on SMT-COMP benchmarks from the *QF_S* and *QF_SLIA* divisions. Results show that s2s is competitive with state-of-the-art solvers OSTRICH [7], Z3 [12], CVC5 [8], and NOODLER [9], and delivers strong performance on both satisfiable and unsatisfiable instances.

II. SOLVER ARCHITECTURE

s2s checks the satisfiability of quantifier-free formulas over strings and string lengths, by translating them into propositional logic and solving the encoding with a SAT solver. Currently, s2s supports formulas over the syntax in Figure 2. In addition to Boolean variables, formulas contain variables of sort *string* and sort *integer*; we denote the sets of string and integer variables by $\mathcal{X}_{str}$ and $\mathcal{X}_{int}$, respectively. The atoms of formulas are word equations, regular constraints, linear integer constraints (over string lengths), and prefix, suffix, and containment constraints.

We give an overview of the architecture of s2s, which is sketched in Figure 1. It consists of three major components, the *Context*, a *Preprocessing* pipeline, and the main *Solving Loop*.

A. Context

The context manages variables and their sort, constants, and formulas. Formulas are represented as *abstract syntax trees* (ASTs). All concrete nodes of the structure are stored in a database, and the AST consists only of reference-counted pointers to these nodes. Leaf nodes are constants (strings,

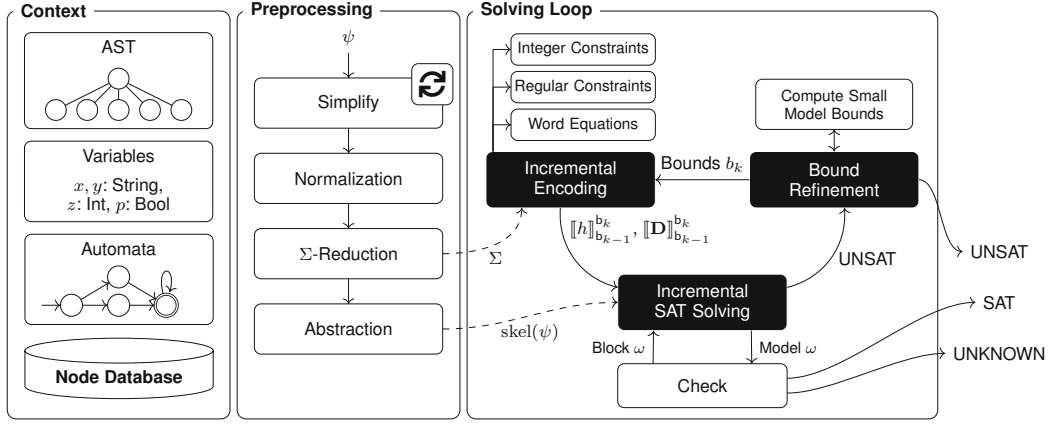


Fig. 1: Architecture of s2s.

$F := F \vee F \mid F \wedge F \mid \neg F \mid A_{str} \mid A_{int} \mid A_{bool}$
 $A_{str} := T_{str} \doteq T_{str} \mid T_{str} \in RE \mid$
 $\quad \text{prefix}(T_{str}, T_{str}) \mid \text{suffix}(T_{str}, T_{str}) \mid \text{contains}(T_{str}, T_{str})$
 $A_{int} := T_{int} \bowtie T_{int} \text{ (with } \bowtie \in \{<, \leq, =, \geq, >\})$
 $T_{str} := T_{str} \cdot T_{str} \mid x \mid w$
 $T_{int} := T_{int} + T_{int} \mid c \cdot T_{int} \mid c \mid |T_{str}| \mid y$

Fig. 2: Supported Syntax. Here $\mathcal{X}_{str}$ and $\mathcal{X}_{int}$ are string and integer variables, RE is a regular expression, A_{bool} are Boolean variables, $x \in \mathcal{X}_{str}$, $y \in \mathcal{X}_{int}$, w is constant string, and $c \in \mathbb{Z}$ is a constant integer.

integers, or regular expressions) or variables, all interned in the node database for reuse. Reference-counted pointers enable constant-time cloning and comparison of nodes. To ensure pointer consistency, all nodes are hash-consed upon creation, guaranteeing that no two syntactically identical nodes exist in the database. As a result, the ASTs are internally represented as DAGs.

Nodes are simplified during construction, e.g., by folding constant terms or resolving simple Boolean equivalences. Additionally, the context provides a facade for compiling regular expressions to finite automata. All conversions are cached (again as reference-counted objects), allowing for constant-time retrieval of repeated conversions of the same regular expression.

B. Preprocessing

Before solving a problem, s2s performs several preprocessing steps. First, it optionally simplifies the formula. Then, it prepares the formula for the central solving loop. This involves the conversion into a normal form, thereby removing unsupported constructs, computing a suitable alphabet for the search procedure, and building the Boolean abstraction.

1) *Simplification*: s2s reduces the search space by running the formula through a pipeline of simplification rules, applied in multiple passes until a fix-point is reached, at which point

no further simplifications are possible. Each pass first applies a series of rewrite rules via a post-order traversal of the AST. For each node, the solver searches for an applicable rewrite rule that replace the node with an equivalent but simpler one. Currently, s2s uses 40 rewrite rules, which are applied in a fixed order. These range from Boolean equivalences, to theory-specific rules, such replacing $\alpha\beta \doteq \alpha\gamma$ with $\beta \doteq \gamma$, or identifying that $a\alpha \doteq b\beta$ is unsatisfiable, where α , β , and γ are string terms and a and b are constant characters. All of these rewrites are standard (see, e.g. [10], [13]).

After the rewrite pass, the simplifier tries to infer variable substitutions that preserve satisfiability when applied to the formula. For example, suppose the formula asserts a word equation of the form $xba \doteq a\beta$ with variable x , constants a, b and arbitrary string terms α, β . The solver detects that necessarily x must start with a and applies the substitution $x \mapsto a \cdot x'$ where x' is a fresh variable. Currently, there are seven rules for inferring substitutions. These rules apply the same reasoning exemplified above to other types of literals, force the value of asserted Boolean variables, or directly set the value of a variable x if it is forced by an asserted equality (e.g., if $x \doteq w$ with $w \in \Sigma^*$, we infer $x \mapsto w$).

The simplifier records all applied substitutions, which are later used for back-substitution when constructing a model for the original formula from a model of the simplified formula.

2) *Normalization*: The SAT encoding is constructed literal-wise, with a corresponding encoding scheme for every type of literal. To minimize the number of encodings, we transform the formula into a *normal form*, which limits of types of literals occurring in the formula. In this form, the formula is in negation-normal form, and each literal is one of the following

- a word equation $\alpha \doteq \beta$ with string terms α, β , or a disequation $x \neq y$ between two string variable $x, y \in \mathcal{X}_{str}$,
- a regular constraint $x \in R$ or $x \notin R$ on a single string variable $x \in \mathcal{X}_{str}$,
- one of $\text{prefix}(w, x)$, $\text{suffix}(w, x)$, or $\text{contains}(x, w)$ where $x \in \mathcal{X}_{str}$ and w a constant string, or
- an integer constraint of the form $\sum_{i=1} t_i \cdot c_i \bowtie c$, where each t_i is either an integer variable or the length of a

string variable, $c, c_i \in \mathbb{Z}$ are constant integers, and $\bowtie$ is one of $\{<, \leq, =, \geq, >\}$.

S2S provides an encoding for each type of literal in normal form. Most formulas in Figure 2 can be rewritten into normal form, possibly by introducing fresh variables. For example, a literal of the form $\alpha \in R$ is rewritten into a formula in normal form $x \in R \wedge x \doteq \alpha$ using a fresh variable x . The only exception are negations of $\text{prefix}(\alpha, \beta)$, $\text{suffix}(\alpha, \beta)$, and $\text{contains}(\alpha, \beta)$ when α is not constant, as they introduce quantification. For example, $\neg \text{prefix}(\alpha, \beta)$ is equivalent to $\forall x. \beta \neq \alpha \cdot x$. If α is a constant string w , then we can model this with a regular constraint $\beta \notin w \cdot \Sigma^*$, where Σ^* accepts any string. Otherwise, these types of literals are not supported and removed from the formula. The formula in normal form is equivalent (modulo new variables) to the original formula if no literals are dropped.

3) *Σ -Reduction*: The SMT-LIB theory of strings [14] pre-defines an alphabet of $3 \cdot 2^{16}$ letters, which is too large for an efficient SAT encoding. Instead, S2S uses the approach of NFA2SAT to construct a subset of this alphabet that preserves satisfiability. The subset consists of all letters occurring in the input problem plus a linear number of extra characters. The exact number of extra characters depends on the formula. If it contains string concatenation, one additional character per inequality [15], otherwise one [11] character per variable.

4) *Abstraction*: In order to handle Boolean combinations, the solver decomposes the input formula ψ into its propositional structure and theory-specific parts, by replacing every (distinct) first-order atom $a \in \text{atoms}(\psi)$ with a fresh Boolean variable p_a and keeps track of the mapping between a and p_a . As a result, we obtain the *Boolean skeleton* $\text{skel}(\psi)$ of ψ and a set $\mathbf{D}$ of *definitions* of the form $a \rightarrow p_a$ ($\neg a \rightarrow \neg p_a$) if a only occurs in positive (negative) polarity in ψ . We call the formula $\text{skel}(\psi) \wedge \bigwedge_{d \in \mathbf{D}} d$ the *Boolean abstraction* and denote it with ψ_A . Based on the results of Plaisted-Greenbaum [16], we have that ψ and ψ_A are equivalent modulo the newly introduced variables in ψ_A .

C. Solving Process

The solving process reduces ψ to a propositional formula $\llbracket \psi \rrbracket$, which is passed to a SAT solver. To construct $\llbracket \psi \rrbracket$, the solver uses a bounded approach: it fixes bounds $\mathbf{b}$: $\mathcal{X}_{str} \cup \mathcal{X}_{int} \rightarrow \mathbb{Z}^2$, where $\mathbf{b}(x) = [l, u]$ denotes the inclusive range of integers $l, \dots, u$ (with $l \geq 0$ for all $x \in \mathcal{X}_{str}$). The encoding ensures that $\llbracket \psi \rrbracket$ is satisfiable precisely if there is a model h of ψ such that $h(x) \in \mathbf{b}(x)$ for all $x \in \mathcal{X}_{int}$ and $|h(x)| \in \mathbf{b}(x)$ for all $x \in \mathcal{X}_{str}$.

Instead of computing precise bounds upfront, S2S starts with small bounds $\mathbf{b}_1$ and gradually increases them. If the encoding $\llbracket \psi \rrbracket^{\mathbf{b}_k}$ is unsatisfiable for some bounds $\mathbf{b}_i$, the solver fixes larger bounds $\mathbf{b}_{k+1}$ and checks the satisfiability of $\llbracket \psi \rrbracket^{\mathbf{b}_{k+1}}$. This results in a series of calls to the SAT solver to determine the satisfiability of the formulas $\llbracket \psi \rrbracket^{\mathbf{b}_1}, \llbracket \psi \rrbracket^{\mathbf{b}_2}, \dots$, each encoding ψ w.r.t. to increasing bounds $\mathbf{b}_1, \mathbf{b}_2, \dots$.

If for some bounds $\mathbf{b}_k$ the SAT solver returns a model ω for $\llbracket \psi \rrbracket^{\mathbf{b}_k}$, then S2S decodes ω into a model h for ψ and checks

if h also solves the original formula (since the preprocessing steps might have removed literals from the original formula). If yes, S2S reports satisfiability, and optionally the model h . If not, then ω is blocked, and S2S searches for another solution within the same bounds $\mathbf{b}_k$. This is repeated until no further model is found within these bounds, in which case the solver gives up and returns *unknown*.

The formula $\llbracket \psi \rrbracket^{\mathbf{b}_k}$ itself is the conjunction

$$\text{skel}(\psi) \wedge \llbracket h \rrbracket^{\mathbf{b}_k} \wedge \llbracket \mathbf{D} \rrbracket^{\mathbf{b}_k}$$

of the Boolean skeleton $\text{skel}(\psi)$ of ψ , an encoding of the set of possible substitutions $\llbracket h \rrbracket^{\mathbf{b}_k}$ over the previously computed alphabet Σ , and encoding of the definitions in $\mathbf{D}$, all bounded by $\mathbf{b}_k$. More precisely, $\llbracket \mathbf{D} \rrbracket^{\mathbf{b}_k}$ is the formula

$$\bigwedge_{(p_a \rightarrow a) \in \mathbf{D}} p_a \rightarrow \llbracket a \rrbracket^{\mathbf{b}_k} \wedge \bigwedge_{(\neg p_a \rightarrow \neg a) \in \mathbf{D}} \neg p_a \rightarrow \llbracket \neg a \rrbracket^{\mathbf{b}_k}$$

where $\llbracket a \rrbracket^{\mathbf{b}_k}$ (resp. $\llbracket \neg a \rrbracket^{\mathbf{b}_k}$) denote the encoding of a literal in normal form a (resp. $\neg a$).

The encoding depends on the type of the literal. Regular constraints $x \in R$ and $x \notin R$ are encoded as the reachability problem of nondeterministic finite automata [11], [17]. For word equations $\alpha \doteq \beta$ and their negations, S2S use the encodings introduced in [15]. The encodings prefix, suffix and containment constraints have been presented in [11]. For linear integer constraint $\sum_{i=1} t_i \cdot c_i \bowtie c$, S2S encodes *multivariate decision-diagrams* similarly to the approach used by WOORPJE [10].

1) *Incremental Encoding*: To make the iterative search process efficient, we rely on *incremental SAT solving under assumptions* [18] to construct the encodings. All our propositional encodings for literals are built incrementally: the encoding of a literal a at bounds $\mathbf{b}_k$ is obtained by only adding clauses to $\llbracket a \rrbracket^{\mathbf{b}_{k-1}}$. More precisely, the encoding ensures $\llbracket a \rrbracket^{\mathbf{b}_k} \Leftrightarrow \llbracket a \rrbracket^{\mathbf{b}_{k-1}} \wedge \llbracket a \rrbracket_{\mathbf{b}_{k-1}}^{\mathbf{b}_k}$ for some set of clauses $\llbracket a \rrbracket_{\mathbf{b}_{k-1}}^{\mathbf{b}_k}$. In the k^{th} call, it is sufficient to generate only the new clauses $\llbracket a \rrbracket_{\mathbf{b}_{k-1}}^{\mathbf{b}_k}$ and add them to the SAT solver, which already retains $\llbracket a \rrbracket^{\mathbf{b}_{k-1}}$ from the $k-1^{\text{th}}$ call. Hence, the encoding $\llbracket \psi \rrbracket^{\mathbf{b}_k}$ is obtained simply by adding clauses to $\llbracket \psi \rrbracket^{\mathbf{b}_{k-1}}$ from the previous iteration.

2) *Bound Refinement*: If $\llbracket \psi \rrbracket^{\mathbf{b}_k}$ is unsatisfiable for some bounds $\mathbf{b}_k$, S2S first checks whether ψ itself is unsatisfiable before updating the bounds. This check relies on the *small-model property* of the theory, which states that every satisfiable formula admits a model of minimal size. If the bounds are large enough to cover such a minimal model, unsatisfiability can be concluded.

To efficiently compute if this is the case, S2S employs the *unsatisfiable core analysis* introduced by NFA2SAT. If an unsatisfiable core C is returned by the SAT solver, S2S constructs a smaller sub-formula ψ^C which is still unsatisfiable but only contains the literals whose encodings are in C . The solver then tries to bound the length of a presumed minimal solution for ψ^C and compares them to $\mathbf{b}_k$. If $\mathbf{b}_k$ covers the bounds of the (presumed) minimal solution for all variables,

TABLE I: Results restricted to the fragment S2S supports. The rows Total, Sat, and Unsat correspond to the number of problems solved in that category. The Timeout row shows the number of problems the solver timed out on. Time is given in seconds. The total time does not include timeouts.

	s2s	CVC5	NOODLER	Z3	OSTRICH
<i>QF_S</i>					
Total	18,782	18,737	18,889	18,663	18,739
Sat	11,973	12,003	12,062	11,982	11,930
Unsat	6,809	6,734	6,827	6,681	6,809
Timeout	135	180	28	254	177
Time Total	694.91	1,034.60	1,506.89	3,762.81	34,579.25
Time SAT	559.12	550.21	1,078.76	2,006.85	23,389.17
Time UNSAT	135.80	484.39	428.13	1,755.96	11,190.08
<i>QF_SLIA</i>					
total	47,895	47,303	48,693	46,615	45,903
sat	24,398	24,072	24,740	23,151	24,098
unsat	23,497	23,231	23,953	23,464	21,805
timeout	896	1,490	80	2,179	2,828
Time Total	4,196.96	7,067.90	2,570.70	9,285.81	118,100.77
Time SAT	3,773.27	5,045.51	1,511.71	8,534.13	51,837.43
Time UNSAT	423.68	2,022.39	1,058.99	751.68	66,263.34
<i>QF_S + QF_SLIA</i>					
total	66,677	66,040	67,582	65,278	64,642
sat	36,371	36,075	36,802	35,133	36,028
unsat	30,306	29,965	30,780	30,145	28,614
timeout	1,031	1,670	108	2,433	3,005
Time Total	4,891.87	8,102.49	4,077.59	13,048.62	152,680.02
Time SAT	4,332.39	5,595.71	2,590.47	10,540.98	75,226.60
Time UNSAT	559.48	2,506.78	1,487.12	2,507.63	77,453.42

then ψ^C has no minimal solution and, therefore, no solution at all. In that case, S2S reports the unsatisfiability of ψ (since $\psi^C \models \psi$). Otherwise, the solver increases the bounds for all variables in $\text{vars}(\psi^C)$, relaxing all intervals $b_k(x)$ by a constant offset, without exceeding the computed bounds on the minimal solution.

If ψ contains no word equations and no integer constraints, then S2S always finds bounds on a minimal solution for all variables (see [11]), making it complete for formulas in this fragment. If the formula contains a word equation or integer constraints, S2S relies on the best-effort heuristic described in [15]. This heuristic may fail to establish bounds on the smallest model, in which case unsatisfiability cannot be detected and S2S will increase variable bounds indefinitely without terminating.

III. EVALUATION

The S2S solver is implemented¹ in Rust and uses CADICAL (version 2.1.3) as the backend SAT solver. It accepts problems in the SMT-LIB format. We have evaluated S2S on the 103,335 problems from the SMT-COMP benchmark suite for the theory of strings². These problems are in two categories: *QF_S* (pure string problems, 18,940 in total), and *QF_SLIA* (string problems with linear integer arithmetic, 84,395 in total).

We compare S2S with four state-of-the-art solvers: CVC5 (version 1.2.1), Z3 (version 4.14.0), OSTRICH (ver-

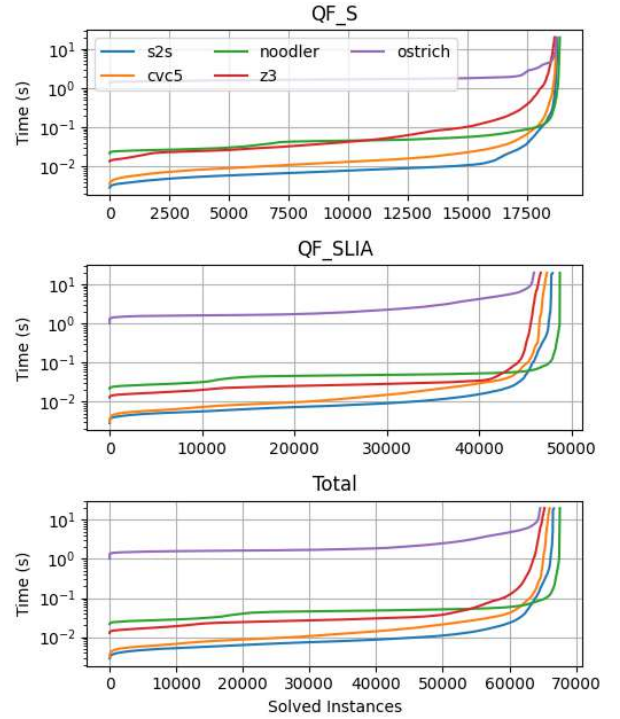


Fig. 3: Cactus plots showing the results on the *QF_S*, *QF_SLIA*, and total benchmarks. Time in seconds (log-scale).

sion 2024-03-22), and NOODLER (version 4.13.0). To ensure soundness, answers from S2S were verified through majority voting, using the other solvers answers as oracles. Additionally, if S2S returned a model for a satisfiable problem, we double-checked the model's correctness by evaluating it on the original formula.

All experiments were executed on an Ubuntu server equipped with 128 cores and 1.7TiB of memory. We used a timeout of 20 seconds per solver and problem, as used in a variety of prior works on string solvers (see e.g., [10], [19]–[21]). We ran 40 solver instances in parallel. No explicit memory limits were imposed, so all instances shared the available system memory.

S2S solves 18,782 from the *QF_S* (99%) and 47,895 from the *QF_SLIA* (57%) problems, respectively, which corresponds to roughly 65% of all problems. On the remaining problems, the solver returned *unknown*, because they fall out of the supported fragment and no solution can be found when removing the literals that cannot be encoded. The evaluation is limited to problems that S2S can solve. Table I summarizes the results and Figure 3 visualizes them as cactus plots.

S2S solves almost all problems in the *QF_S* category and is the fastest solver overall in that fragment. Only CVC5 solved more satisfiable problems in less time, but it solved fewer problems in total. While NOODLER solved more problems than S2S, S2S was considerably faster, particularly on unsatisfiable instances. Both Z3 and OSTRICH solved fewer problems than S2S and took substantially more time. Notably, S2S was much faster on unsatisfiable problems than any other solver.

¹ Available at <https://github.com/s2ssolver/s2s>

² Benchmarks are available at <https://zenodo.org/records/11061097>

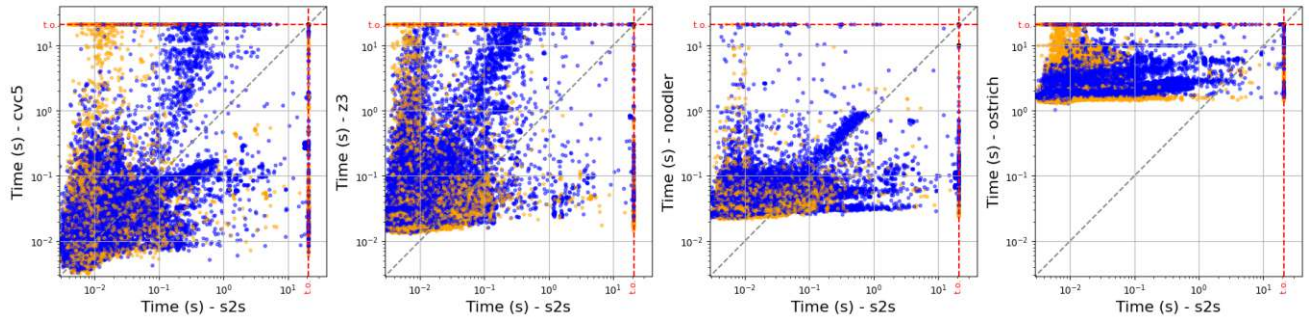


Fig. 4: Scatter plots comparing the runtime of S2S with other solvers on all ($QF_S + QF_SLIA$) problems. The axes show the runtime in log-scale. Blue dots are satisfiable and orange dots are unsatisfiable instances. The dashed red lines indicate timeouts.

Restricting to the problems that S2S can solve, we see similar results on the QF_SLIA set. In that fragment, S2S is almost twice as fast as CVC5 and solves 492 more problems within the time limit. Among all solvers, NOODLER solved the most problems and was the fastest overall. On satisfiable problems, NOODLER was almost twice as fast as S2S, which ranked second. On unsatisfiable problems, the pattern reversed: S2S was twice as fast as NOODLER, though NOODLER solved considerably more of them. As with the QF_S problems, Z3 and OSTRICH solved fewer problems and took more time than the other solvers.

The scatter plots in Figure 4 give a more detailed comparison between the solvers. They compare the runtimes of S2S (x-axis) to the runtimes of CVC5, Z3, NOODLER, and OSTRICH, respectively (y-axis). Every dot represents a single problem, with the axes indicating the runtime of each solver for that problem. Blue dots correspond to satisfiable problems, while yellow dots indicate unsatisfiable ones. A dot is above the diagonal precisely if S2S solved the problem faster. We can see that, independently of the solver we compare to, most orange dots are above the diagonal. This aligns with the findings that S2S is generally fast on unsatisfiable instances.

We can see that almost all solvers need a constant amount of time before solving any problem, visible as a gap between the axis and the lowest data point. We refer to this as the *startup time*. The solvers Z3, NOODLER (which is based on Z3), and OSTRICH have relatively high startup time, compared to S2S, which show little to no startup time. CVC5 has only a slightly higher startup than S2S and for a small subset that CVC5 solves fast, it has little to no startup time at all. We reckon that these might be instances that can be solved merely by simplification, without starting the search procedure.

IV. CONCLUSION

We presented S2S, an eager string solver that reduces string constraints to propositional logic and solves them via incremental SAT solving. The solver combines the ideas of NFA2SAT and WOORPJE. Despite supporting only a fragment of the full SMT-LIB theory, S2S achieves strong performance on a large fraction of the SMT-COMP benchmarks. Unlike

lazy solvers, S2S avoids dedicated theory engines and offers an alternative for the supported fragment.

We plan to extend expressiveness by support to additional types of literals, such as transduction. This requires finding efficient SAT encodings that integrate with the incremental solving approach. Another possible research direction is to explore more sophisticated CEGAR approaches. Using appropriate over-approximations of the input formula could lead to more efficient SAT encodings and significantly improve the solver’s performance. Finally, we believe that bound refinement procedure can be improved through careful in-processing, likely resulting in further performance improvements.

REFERENCES

- [1] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin, *String Analysis for Software Verification and Security*. Springer International Publishing, 2017. [Online]. Available: <http://dx.doi.org/10.1007/978-3-319-68670-7>
- [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)*, 2018, pp. 1–9.
- [3] G. S. Makanin, “The problem of solvability of equations in a free semigroup,” *Mathematics of the USSR-Sbornik*, vol. 32, no. 2, pp. 129–198, Feb. 1977. [Online]. Available: <http://dx.doi.org/10.1070/SM1977v032n02ABEH002376>
- [4] K. U. Schulz, “Makanin’s algorithm for word equations—two improvements and a generalization,” in *Word Equations and Related Topics*, K. U. Schulz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 85–150.
- [5] M. Berzish, J. D. Day, V. Ganesh, M. Kulczynski, F. Manea, F. Mora, and D. Nowotka, “String theories involving regular membership predicates: From practice to theory and back,” in *Combinatorics on Words*, T. Lecroq and S. Puzynina, Eds. Cham: Springer International Publishing, 2021, pp. 50–64.
- [6] J. D. Day, V. Ganesh, P. He, F. Manea, and D. Nowotka, “The satisfiability of word equations: Decidable and undecidable theories,” in *Reachability Problems*, I. Potapov and P.-A. Reynier, Eds. Cham: Springer International Publishing, 2018, pp. 15–29.
- [7] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, “Decision procedures for path feasibility of string-manipulating programs with complex operations,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290362>
- [8] H. Barbosa, C. 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 *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 415–442.

- [9] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, “Z3-noodler: An automata-based string solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 24–33.
- [10] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, “On solving word equations using SAT,” in *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*, ser. Lecture Notes in Computer Science, E. Filiot, R. M. Jungers, and I. Potapov, Eds., vol. 11674. Springer, 2019, pp. 93–106. [Online]. Available: https://doi.org/10.1007/978-3-030-30806-3_8
- [11] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka, “Solving string constraints using SAT,” in *Computer Aided Verification*, C. Enea and A. Lal, Eds. Cham: Springer Nature Switzerland, 2023, pp. 187–208.
- [12] L. de Moura and N. Björner, “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.
- [13] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli, “Scaling up DPLL (T) string solvers using context-dependent simplification,” in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 453–474.
- [14] C. Tinelli, C. Barrett, and P. Fontaine, “SMT-LIB theory of unicode strings,” 2020. [Online]. Available: <https://smt-lib.org/theories-UnicodeStrings.shtml>
- [15] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, and D. Nowotka, “Solving string constraints with concatenation using SAT,” in *Formal Methods in Computer-Aided Design*, N. Narodytska and P. Rümmer, Eds. TU Wien Academic Press, 2024.
- [16] D. A. Plaisted and S. Greenbaum, “A structure-preserving clause form translation,” *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, 1986. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0747717186800281>
- [17] M. Kulczynski, K. Lotz, D. Nowotka, and D. B. Poulsen, “Solving string theories involving regular membership predicates using SAT,” in *Model Checking Software*, O. Legunsen and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 134–151.
- [18] N. Eén and N. Sörensson, “Temporal induction by incremental SAT solving,” *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, 2003, bMC’2003, First International Workshop on Bounded Model Checking. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1571066105825423>
- [19] M. Berzish, V. Ganesh, and Y. Zheng, “Z3str3: A string solver with theory-aware heuristics,” in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 55–59.
- [20] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh, “Z3str4: A multi-armed string solver,” in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 389–406. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-6_21
- [21] M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, “The power of string solving: Simplicity of comparison,” in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, ser. AST ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 85–88. [Online]. Available: <https://doi.org/10.1145/3387903.3389317>

FastPoly: An Efficient Polynomial Package for the Verification of Integer Arithmetic Circuits

Alexander Konrad Christoph Scholl
University of Freiburg, Freiburg, Germany
{konrada, scholl}@informatik.uni-freiburg.de



Abstract—In recent years, methods based on Symbolic Computer Algebra (SCA) have become increasingly successful in the field of formal verification of arithmetic circuits. While several different approaches have been proposed to tackle this challenging task, most of them are based on the same mathematical operations. They perform ideal membership tests that reduce specification polynomials by a series of polynomial divisions. For integer arithmetic, under certain conditions, the polynomial divisions boil down to substitutions of variables in integer polynomials. In this context, the overall performance of an SCA verification tool is closely related to the efficiency of the steps manipulating integer polynomials. In this paper we present our tool FastPoly, a package for representing integer polynomials that provides efficient operations including variable substitution with integrated normalization steps. We provide the sources of our tool, making it available for other research groups to support future progress in this field.

I. INTRODUCTION

Arithmetic circuits play an important role in circuit designs, ranging from general-purpose processors to specialized hardware used in computationally intensive applications such as cryptography or machine learning. In order to increase the confidence in the designs and to avoid design errors such as the infamous Pentium bug [1], fully automatic formal verification became more and more important.

In particular, the verification of multiplier and divider circuits has been a challenging problem for a long time. Methods based on BDDs [2], [3] struggle with exponential space complexity while SAT-based methods [4], [5] experience exponential run times. However, methods based on *Symbolic Computer Algebra* (SCA) have become increasingly successful in recent years, enabling the verification of large and complex arithmetic circuits like finite field multipliers [6], integer multipliers [7]–[23], modular multipliers [24] and divider circuits [25]–[29]. Here the verification task has been reduced to an ideal membership test for the specification polynomial based on so-called backward rewriting, sometimes also referred to as algebraic reasoning [19], [20], [22], [30].

In the context of SCA-based formal verification for integer arithmetic, a variety of different approaches have been published recently. Some focus on reverse engineering and detection of converging cones to precompute polynomials for sub-circuits and simplify those polynomials early on [15], [16], [18], [21]. Some rely on adder detection to simplify the circuit under verification and additionally split the ideal membership

test into several sub-tasks by splitting the specification into “slices”, one for each primary output [17], [19], [20], [22], [30]. Others precompute specific signal relations in the circuit like equivalences/antivalences [27] or satisfiability don’t cares [28], [29] to simplify occurring intermediate polynomials which especially is necessary for divider verification. One of the most recent methods showed that a successful verification of integer multipliers can be achieved by only using dynamic phase and order optimization [23].

Despite differences in the way their algorithms work, at the lowest level all of these approaches rely on the same basic operations: Representing specification polynomials and performing manipulation steps on them, either by polynomial divisions or the simpler substitutions of variables by integer polynomials. In the remainder of the paper we will refer to those manipulations as *substitution steps*. Those substitution steps influence the higher-level algorithms in several ways: First, the overall performance of the algorithm is closely related to the efficiency of single substitution steps. This is easy to conclude, because each approach ultimately boils down to a certain number of necessary substitution steps that it must perform. Second, the efficiency of single substitution steps already influences the design options of the algorithms. If a single substitution step already has high computational costs, algorithmic approaches reducing the number of substitution steps or splitting the specification polynomial into smaller slices become more important and elaborate methods searching for good substitution orders to reduce peak sizes of polynomials as used in [23] become infeasible.

In this paper we present FastPoly, an integer polynomial package that provides efficient operations on polynomials including the substitution steps which are the basis for many SCA-based algorithms. We elaborate how we achieve an efficient implementation of those substitution steps by using two simple data structures. Our experimental results show the efficiency of our package in contrast to other publicly available polynomial packages which provide a similar operation set. By providing the sources of our package and making them available for other research groups, we encourage and support future progress in this field.

The paper is structured as follows: In Sect. II we provide details on our polynomial package. In Sect. III we give a demonstration of how to use our package. We evaluate our

package in Sect. IV and conclude the paper with final remarks in Sect. V.

II. TOOL DESCRIPTION

A. Requirements

Before diving into the details of our polynomial package, we first summarize the requirements for a polynomial package that is suitable to be used in SCA-based approaches to the verification of integer arithmetic circuits.

We start by defining the polynomials we want to represent as well as their basic characteristics: 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 monomials, a monomial is a product of a term with an integer, and a term is a product of variables from X . Polynomials represent *pseudo-Boolean functions* $f : \{0,1\}^n \mapsto \mathbb{Z}$. The very basic operation that our polynomial package has to provide (in an efficient way) is the substitution step of the SCA-based backward rewriting approach. In this step all occurrences of a distinct variable x_i in the polynomial are replaced by some other polynomial P_i , i.e., the substitution with $x_i = P_i$ (or the polynomial division with the polynomial $-x_i + P_i$) is performed. To achieve canonical representations, polynomials also have to be simplified after each substitution step by reducing powers v^k of variables v with $k > 1$ to v (since the variables are binary), by combining monomials with identical terms into one monomial, and by omitting monomials with leading factor 0. For those normalization steps we need the functionalities of finding, adding and removing monomials. For example, if we want to add a monomial $c \cdot m$ (which originates from a substitution step or has to be added for other reasons) to our polynomial, we first have to search for a monomial $d \cdot m$ that already exists in the polynomial. In this case, the old $d \cdot m$ is removed and the new monomial $(c + d) \cdot m$ (if $c + d \neq 0$) is added to the polynomial. In this way we make sure that there is only up to one monomial with the same term in the polynomial. In the context of circuit verification, x_i typically represents the output of a circuit component and P_i is the polynomial description of the function of this component expressed in its input variables. Such components can be basic logic gates such as AND and OR gates with the corresponding polynomials being $P_i = a_i \cdot b_i$ and $P_i = a_i + b_i - a_i \cdot b_i$, respectively, but they can also describe larger parts of the circuit, such as fanout-free cones, as used in [16]. Fig. 1 shows the series of substitution steps for a full adder circuit.

In summary, our polynomial package has to represent and normalize polynomials as described above. We need to be able to find monomials in the polynomial quickly, we have to add and remove monomials quickly, and we need an efficient way of carrying out the basic substitution step. Next we describe how we achieve those goals in our polynomial package FastPoly.

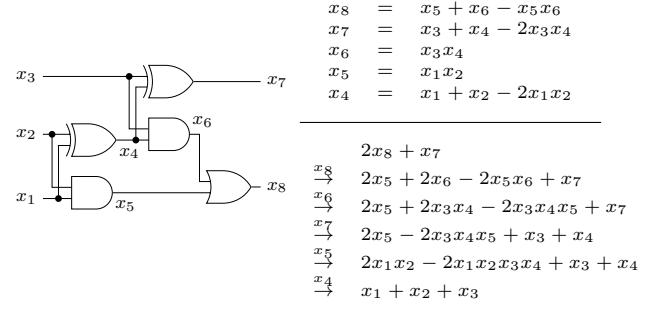


Fig. 1. Full adder circuit with series of substitutions.

B. Implementation

Our polynomial package FastPoly is implemented in C++. In this subsection we break down the data structures used and how they support the necessary operations.

1) *Polynomials*: Our class for polynomials, called *Polynom*, is the data structure saving all information needed for the representation of integer polynomials and enabling an efficient implementation of the substitution operation. The class consists of two major parts. The first part uses the `std::set` from C++ for saving all monomials of a polynomial. We elaborate on the monomial class in the next subsection. For now it is only important to know that we have monomial objects representing monomials and those monomials have a specific order defined among each other which is used to determine a monomial's position in the set. The `std::set` is usually implemented as red-black tree. Therefore, searching, inserting and removing monomials have logarithmic time complexity in the size of the set, i.e., the size of the polynomial.

However, for the desired substitution operation, where we substitute all occurrences of a variable x_i in the polynomial by some other polynomial P_i , this set is not sufficient to enable an efficient implementation of the operation. As mentioned, searching for a monomial has logarithmic time complexity, which is good enough for our needs as long as we are searching for a single specific monomial. In contrast, for the substitution operation we do not search for a single monomial only, but we have to find all monomials which include the variable x_i we want to substitute. To avoid the necessity of searching through the complete polynomial for this purpose, we use a second index data structure for our polynomial class which we call *refList*. Basically, we maintain for every variable in the polynomial a doubly linked list. The elements stored in the doubly linked list for some variable x_i are pointers to all monomials containing x_i . The heads of the doubly linked lists are stored in a C-style array. Now, for the substitution operation we find all monomials containing a specific variable x_i by obtaining (in constant time) the head of the list for x_i from the array under index i and by following the doubly linked list afterwards. If a variable is not present in the polynomial, the corresponding list will be empty, which can be checked in constant time as well.

2) *Monomials*: Now, we define the implementation of monomials in detail. Our class for monomials is called *Monom* and consists of the following: The coefficient is realized by an arbitrary-precision integer implemented by the GNU Multiple Precision Arithmetic Library (GMP) [31]. A usual 64-bit integer is not sufficient here since in circuit verification we typically deal with very large coefficients. The term of the monomial is a C-style array of integers which is always sorted in ascending order. An integer i represents the variable x_i . We use an additional integer to save the size of the array and an additional integer to save the sum over all indices of variables in the monomial. This sum can be seen as a very light-weight form of a hash value for each monomial which is used to differentiate between monomials more quickly. For saving the monomials into the `std::set` of the polynomial class, we have to define a specific order on monomials. To compare two monomials, we first compare the sum components of the monomials. In case the sums are identical, we compare the sizes of the monomials next. If the sizes are identical as well, we fall back to a lexicographic order of the terms of the compared monomials. For this lexicographic order, we may need, in the worst case, to compare all variables in both terms of the compared monomials. For this reason we compare the sum and size attributes of monomials first, since most monomials differ already either in their sum or their size. Note that the coefficients of monomials are *not* used for ordering. There is only up to one monomial with the same term in a polynomial and during the addition of a monomial $c \cdot m$ to a polynomial p we just look for *some* existing monomial $d \cdot m$ in p with arbitrary coefficient d . Additionally, the representation of a monomial m contains for each variable x_i a “back” pointer to the element in the doubly linked list for x_i that points to m . In that way, m can be removed from the index data structure formed by the doubly linked lists in time that is linear in the size of m . Fig. 2 provides a visualization of the explained data structures for the simple example polynomial $P = 5x_1 + 7x_2 + 9x_1x_3$.

C. Additional Features

In addition to the basic structure of our polynomial package explained in the previous subsection, we provide the following features:

- **Internal modulo reduction**: Our package provides the possibility to set a modulo reduction parameter, enabling an internal modulo operation on monomial coefficients already during internal polynomial operations. This ensures that all monomial coefficients are calculated modulo the reduction parameter, which is used in many SCA-based approaches [23].
- **Phase Optimization**: We offer the functionality to change the “phase” of a variable in the polynomial by replacing each occurrence of the variable by its negation. This can be used to perform phase optimization with the goal to reduce the polynomial size [23].
- **Certification**: We offer the ability to produce certificates in the practical algebraic calculus (PAC) format [32].

III. TOOL USAGE

We provide the source code of our tool at [33]. Our tool relies on the GMP library [31] which has to be preinstalled by the user. We currently provide three different APIs for the usage of our polynomial package (which also can be used in a mixed fashion):

- 1) Reading in the starting polynomial and the polynomials for the substitution steps from an external file.
- 2) Building polynomials from scratch using the constructors of the polynomial and monomial classes and performing substitution steps of variables by polynomials by the basic substitution function.
- 3) Using pre-defined “shortcut” functions for substitution steps of basic logic gates.

In Listing 1 we provide a demonstration on how to use these APIs for the simple example of backward rewriting a full adder circuit which is shown in Fig. 1.

IV. EXPERIMENTAL RESULTS

We have tested the performance of our package on a number of sample cases. The tests were executed on a single core of an Intel Xeon CPU E5-2643 with 3.30 GHz. Resources were limited to 32 GB of main memory and 6 hours of CPU time. For comparison we ran the same experiments for the general-purpose computer algebra system SINGULAR [34] as well as for the polynomial library used in AMULET2.2 [30], since it is, to the best of our knowledge, the only other publicly available polynomial package for the specific purpose of SCA-based verification.

We provide an artifact at [35] for better reproducibility of our experimental results.

For the benchmark set we decided to use some examples from SCA-based multiplier verification [23]. Since we want to evaluate the performance of our presented package and compare it to the polynomial library from AMULET2.2 and to the computer algebra system SINGULAR, we abstract from differences in the application of the polynomial packages in our verification tool DYNPHASEORDEROPT [23] and in AMULET2.2. We achieve this by using benchmarks which consist only of a starting polynomial (which is equal to the specification polynomial from [23]) and a (pre-computed) series of substitution steps. To create these benchmarks, we use DYNPHASEORDEROPT which intensively searches for a “good” traversal order (in terms of small intermediate polynomial sizes) of the gates of a given multiplier circuit. It achieves this by determining candidates for the next substitutions, trying out these substitution steps and, if the polynomial size grows too large, reverting the steps again until a suitable series of substitutions is found. We write out the resulting substitution order from DYNPHASEORDEROPT to be able to “replay” it later on. In summary, the experiments we consider here consist only of construction and substitution of polynomials, with the substitution steps predetermined in the benchmark files. Note that DYNPHASEORDEROPT actually performs an additional “phase optimization” after every step. The phase

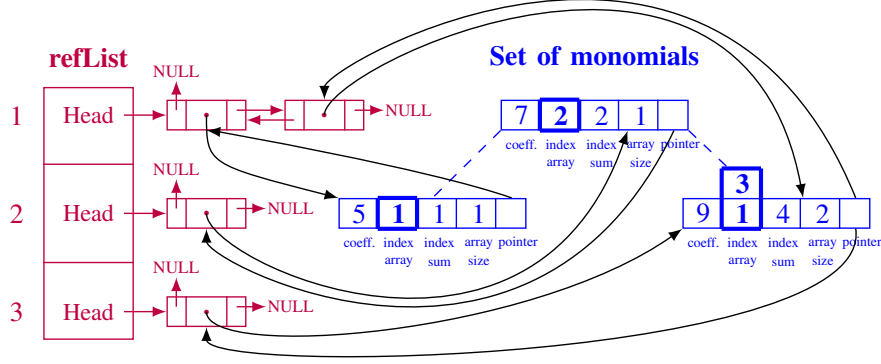


Fig. 2. Visualization of our polynomial data structures.

TABLE I
EXPERIMENTAL RESULTS. TIMES IN CPU SECONDS.

Benchmark	# steps	Starting poly.size	Run times	Run times	Run times	Maximum poly.size
			FastPoly	AMULET2.2	Singular	
16-bit sp-ar-rc	2,816	288	0.85	0.82	TO	302
32-bit sp-ar-rc	11,776	1,088	2.53	9.08	TO	1,102
64-bit sp-ar-rc	48,128	4,224	9.69	103.58	EE	4,298
128-bit sp-ar-rc	194,560	16,640	38.77	7444.54	EE	16,654
256-bit sp-ar-rc	782,336	66,048	150.35	TO	EE	66,062
16-bit sp-wt-lf	3,057	288	0.58	1.24	TO	670
32-bit sp-wt-lf	12,616	1,088	2.41	20.97	TO	2,055
64-bit sp-wt-lf	50,564	4,224	9.84	455.97	EE	8,315
128-bit sp-wt-lf	200,100	16,640	38.65	TO	EE	26,522
256-bit sp-wt-lf	796,191	66,048	158.00	TO	EE	82,154
8-bit sp-ar-ks	652	80	1.77	TO	TO	54,473

optimization is omitted, since the polynomial library from AMULET2.2 does not support this operation. We considered three different multiplier architectures: sp-ar-rc, sp-wt-lf, and sp-ar-ks [36]. Usually, 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. In our considered architectures this is realized by a simple PPG (sp), which just computes the logical AND of all bits of the first input and all bits of the second input. The second stage is the Partial Product Accumulator (PPA) which sums up all the partial products until they are reduced to two numbers only. In our experiments we consider array accumulation (ar) and Wallace trees (wt) [37]. 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, realized by a two operand adder network. We consider the well-known ripple-carry adder (rc), the Ladner-Fischer adder (lf) [38] and the Kogge-Stone adder (ks) [39]. The former two architectures sp-ar-rc and sp-wt-lf can be considered as easy to verify and lead to rather low peak sizes of the polynomials when choosing a reasonable substitution order. The latter, sp-ar-ks, is more challenging for SCA-based verification tools. Therefore, we consider bitwidths ranging from 16 to 256 bits for the two easy architectures, while we only consider the 8-bit benchmark for sp-ar-ks. We

would like to point out that we have deliberately chosen a few architectures as examples for which we know that the computed series of substitution steps does not produce excessively large intermediate polynomials, even without phase optimization. A comprehensive experimental analysis showing the robustness of our verification tool DYNPHASEORDEROPT on different multiplier architectures can be found in [23]. For example, the successful verification of challenging circuits like the sp-ar-ks multipliers for much larger bitwidths than eight (as considered here) definitely needs the dynamic phase optimization from [23] as well.

The experimental results are shown in Tab. I. Col. 1 states the benchmark. Col. 2 gives the number of substitution steps for this benchmark and Col. 3 the size of the starting polynomial. Col. 4 indicates the run time for our package to execute all substitution steps while Col. 5 gives the run time for the polynomial library from AMULET2.2 [30]. Col. 6 shows the results for the computer algebra system Singular. Col. 7 shows the maximum polynomial size reached by intermediate results for our package as well as for AMULET2.2. Since we perform the same series of substitutions for both packages, they produce the same intermediate results leading to identical maximum polynomial sizes. However, we cannot confirm this for SINGULAR, because it does not provide intermediate results.

Listing 1. Full adder demo.

```
#include "poly_parser.h"

int main(int argc, char ** argv) {

    // Example 1) Reading in from external file.
    Polynom spec1;
    init_spec(spec1, ".demo/fulladder_example.txt");
    reduce_poly(spec1, ".demo/fulladder_example.txt");
    std::cout << "Result1:" << spec1 << std::endl;

    //Example 2) Building polynomials from scratch.
    Polynom x8poly(6); // Instantiate poly for x8 gate.
    // The initialization parameter defines the maximum
    // variable index for the polynomial.
    x8poly.createMonom(5); // Create and add monom 1*x5.
    x8poly.createMonom(6); // Create and add monom 1*x6.
    x8poly.createMonom(5,6, mpz_class(-1));
    // Create and add monom -1*x5*x6.
    Polynom x7poly(4); // Instantiate poly for x7 gate.
    x7poly.createMonom(3);
    x7poly.createMonom(4);
    x7poly.createMonom(3,4, mpz_class(-2));
    Polynom x6poly(4); // Instantiate poly for x6 gate.
    x6poly.createMonom(3,4);
    Polynom x5poly(2); // Instantiate poly for x5 gate.
    x5poly.createMonom(1,2);
    Polynom x4poly(2); // Instantiate poly for x4 gate.
    x4poly.createMonom(1);
    x4poly.createMonom(2);
    x4poly.createMonom(1,2, mpz_class(-2));
    Polynom spec2(8); // Instantiate spec2 polynomial.
    spec2.createMonom(8, mpz_class(2));
    spec2.createMonom(7);
    // Now substitute xi by its gate polynomial xipoly.
    spec2.replaceVarByPoly(8, x8poly);
    spec2.replaceVarByPoly(6, x6poly);
    spec2.replaceVarByPoly(7, x7poly);
    spec2.replaceVarByPoly(5, x5poly);
    spec2.replaceVarByPoly(4, x4poly);
    std::cout << "Result2:" << spec2 << std::endl;

    //Example 3) Using "shortcut" logic gate functions.
    Polynom spec3(8); // Instantiate spec3 polynomial.
    spec3.createMonom(8, mpz_class(2));
    spec3.createMonom(7);
    spec3.replaceOR(8, 5, 6);
    // Function for replacing x8 by the OR gate
    // polynomial with inputs x5 and x6.
    spec3.replaceAND(6, 3, 4);
    spec3.replaceXOR(7, 3, 4);
    spec3.replaceAND(5, 1, 2);
    spec3.replaceXOR(4, 1, 2);
    std::cout << "Result3:" << spec3 << std::endl;

    return 0;
}
```

It can be seen that our package clearly outperforms the polynomial library from AMULET2.2 and SINGULAR. This can be explained by the fact that in the polynomial library from AMULET2.2 the performance of a single substitution step clearly depends on the size of the current polynomial, even if only a fraction of the polynomial is actually changed. This resulted in long run times and even time outs (indicated by TO) for benchmarks which produce large intermediate polynomials like the 128-bit and 256-bit examples as well as the challenging 8-bit sp-ar-ks benchmark. SINGULAR was not able to solve any of the benchmarks. The smaller benchmarks ran into time outs (TO), while benchmarks with bitwidths of 64 and larger produced an error state (indicated by EE) because SINGULAR cannot handle more than 32767 ring variables. This result

was to be expected. Although SINGULAR, as general-purpose computer algebra system, provides the necessary operations to perform the polynomial substitution steps, it is not specifically tailored for our considered use case, unlike our polynomial package as well as the one from AMULET2.2. Our package, on the other hand, was able to solve every benchmark within a few minutes. We can observe that the run times of our package increase roughly linearly with the number of substitution steps (which in turn grow quadratically with the bitwidth of the benchmark). We would like to emphasize that the efficiency of the substitution steps shown for our polynomial package is key to enabling the intensive order calculations which are performed in our verification tool from [23].

V. CONCLUSIONS AND FUTURE WORK

We have presented our tool FastPoly, a polynomial package designed for supporting efficient polynomial operations which are used in SCA-based approaches for circuit verification. Our experimental results confirm the efficiency of our tool. We believe by making our tool available for other research groups we will support future advances in the field of SCA-based circuit verification.

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] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.
- [13] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on And-Inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.
- [14] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.
- [15] 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.
- [16] —, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.
- [17] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.

- [18] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.
- [19] D. Kaufmann and A. Biere, "AMulet 2.0 for verifying multiplier circuits," in *TACAS*. Springer, 2021, pp. 357–364.
- [20] D. Kaufmann, P. Beame, A. Biere, and J. Nordström, "Adding dual variables to algebraic reasoning for gate-level multiplier verification," in *DATE*. IEEE, 2022.
- [21] 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.
- [22] 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.
- [23] A. Konrad and C. Scholl, "Symbolic computer algebra for multipliers revisited - it's all about orders and phases," in *FMCAD*. IEEE, 2024, pp. 261–271.
- [24] 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.
- [25] 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.
- [26] —, "Functional verification of hardware dividers using algebraic model," in *VLSI-SoC*, 2019, pp. 257–262.
- [27] C. Scholl and A. Konrad, "Symbolic computer algebra and SAT based information forwarding for fully automatic divider verification," in *DAC*, 2020.
- [28] 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.
- [29] 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.
- [30] D. Kaufmann and A. Biere, "Fuzzing and delta debugging and-inverter graph verification tools," in *TAP@STAF*. Springer, 2022, pp. 69–88.
- [31] T. Granlund and the GMP development team, "GNU MP: The GNU Multiple Precision Arithmetic Library," 2023, <https://gmplib.org/>.
- [32] 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.
- [33] A. Konrad and C. Scholl, "FastPoly source code," 2025. [Online]. Available: <https://github.com/a-konrad/fastpoly>
- [34] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, "SINGULAR 4-2-1 — A computer algebra system for polynomial computations," <https://www.singular.uni-kl.de>, 2021.
- [35] A. Konrad and C. Scholl, "Artifact for FastPoly: An efficient polynomial package for the verification of integer arithmetic circuits," 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.16744818>
- [36] 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.
- [37] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Comp.*, vol. EC-13, pp. 14–17, 1964.
- [38] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.
- [39] 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.

OSTRICH2: Solver for Complex String Constraints

Matthew Hague 

Department of Computer Science
Royal Holloway, University of London
matthew.hague@rhul.ac.uk

Denghang Hu 

Key Laboratory of System Software
Institute of Software
Chinese Academy of Sciences
hudh@ios.ac.cn

Artur Jeż 

Institute of Computer Science
University of Wrocław
aje@cs.uni.wroc.pl

Anthony W. Lin 

University of Kaiserslautern-Landau
Max-Planck Institute for Software Systems
awlin@mpi-sws.org

Oliver Markgraf 

University of Kaiserslautern-Landau
markgraf@cs.uni-kl.de

Philipp Rümmer 

University of Regensburg
Uppsala University
philipp.ruemmer@ur.de

Zhilin Wu 

Key Laboratory of System Software
Institute of Software
Chinese Academy of Sciences
wuzl@ios.ac.cn



Abstract—We present OSTRICH2, the latest evolution of the SMT solver OSTRICH for string constraints. OSTRICH2 supports a wide range of complex functions on strings and provides completeness guarantees for a substantial fragment of string constraints, including the straight-line fragment and the chain-free fragment. OSTRICH2 provides full support for the SMT-LIB theory of Unicode strings, extending the standard with several unique features not found in other solvers: among others, parsing of ECMAScript regular expressions (including look-around assertions and capture groups) and handling of user-defined string transducers. We empirically demonstrate that OSTRICH2 is competitive to other string solvers on SMT-LIB benchmarks.

I. INTRODUCTION

Strings are ubiquitous in modern software systems, especially with the advent of programming languages like JavaScript, PHP, and Python. Despite this, string manipulation is well-known to be error-prone and can easily lead to security vulnerabilities including HTML injection (e.g. see [1], [2]). Applications to analysis of security vulnerabilities caused by string manipulation have been one of the main catalysts for the extensive research into SMT over strings spanning across the last twenty years [2], [3], [4], [5], [6], [7], [8]. One of the success stories of string solvers includes their usage at AWS for analysis of Role-Based Access Control (RBAC) policies (e.g. see [9], [10]). Since 2020 SMT-LIB Unicode String theory [11] has been formalized and supported by many existing string solvers, including Z3 [12], Z3-alpha [13], Z3-Noodler [14], [15], [16], [17], Z3str3RE [18], cvc5 [19], Z3str4 [20], Trau [21], [22], [23], and our own solver OSTRICH [4].

In this paper, we present OSTRICH2, the latest evolution of the OSTRICH string solver [4]. The primary contribution

of this paper is a complete and unified description of OSTRICH2's architecture, algorithms, and implementation as a modular and extensible system. Over the years, OSTRICH has undergone extensive internal development such as integrating new solving techniques, heuristics, and architectural changes, but many of these advances have not been previously documented or published. This includes, for instance, the use of word equation splitting strategies, character-count and length abstractions in preprocessing, and a portfolio-based orchestration of multiple solving engines. OSTRICH2 also includes a newly introduced solver based on algebraic data types (ADT-Str), described here for the first time. This paper is the first to present the overall system design of OSTRICH2, detailing how its components interact within a modular proof framework. We also provide the first full account of OSTRICH2's preprocessing strategies and rule infrastructure. These advances have radically enhanced the solver and resulted in significant performance improvements; notably, OSTRICH achieved first place in the QF_S track of the SMT-COMP 2023 competition. Together, this constitutes the first comprehensive system description of OSTRICH2. To this end, rather than being exhaustive, we aim for accessibility by taking the reader through illustrative examples (among others). In addition to unravelling the design of OSTRICH2, we also report our latest experimentation with the solver on SMT-LIB'25 benchmarks, showing its competitiveness, most notably on unsatisfiable instances.

The following is a list of main features of OSTRICH2:

- (F1) SMT-LIB Unicode String theory inputs.
- (F2) Native support of features of ECMAScript regular expressions (e.g., lookaround assertions, capture

groups, and references; back-references are supported only in the replacement string of `replace` and `replace_all`, not in the matching expression).

- (F3) Native support of `replace_all` and, more generally, complex string transductions using a more comprehensive *regular constraint propagation* strategy, which *combines* backward and forward regular propagations for a plethora of complex functions.
- (F4) An extensive portfolio of solving strategies that includes a cost-enriched string solver (CE-Str) [24], a list-based ADT solver (ADT-Str), and preprocessings (length/character-count abstraction).

Related Tools.: Over the past decade, numerous string solvers have been proposed, employing techniques such as bit-vector encodings, automata-based propagation, and reductions to word equations. Several early or now-unsupported tools do not handle SMT-LIB 2.6 and are no longer actively maintained, including HAMPI [6], Kaluza [25], STRANGER [26], S3 [27], Norn [3], Trau [23], and members of the Z3Str family such as Z3str3 [28] and Z3str3RE [18]. These solvers contributed important techniques such as word-equation reasoning with advanced heuristics and direct regular expression support, but have not appeared in recent SMT-COMP competitions and are not included in our evaluation.

Our comparisons focus on state-of-the-art, actively maintained SMT solvers with SMT-LIB 2.6 support. Z3-alpha [13], which builds on Z3str4 [20] and Z3 [12], remains actively maintained and incorporates strategy synthesis over multiple string-solving backends. Z3 [12] and cvc5 [19] combine rewriting, word-equation decomposition, and regular constraint reasoning. Z3-Noodler [14] introduces a stabilization algorithm that propagated information on regular constraints to a selected word equation until the inferred regular languages on both sides stabilize. The solver is complete for the chain-free fragment but does not support string transductions or `replace_all`.

The original OSTRICH solver [4] implemented a backward regular constraint propagation strategy, providing completeness for the straight-line fragment and support for transductions. Subsequent extensions added the cost-enriched solver CE-Str [24] and ECMAScript-style regex support [29]. OSTRICH2, described in this paper, is the first to combine these capabilities with additional preprocessing, a newly implemented ADT-based solver, and a portfolio orchestration of multiple engines, as detailed in Section III.

Organization.: Section II reviews the SMT-LIB Unicode Strings standard and our grammar; Section III reviews the architecture of OSTRICH2; Section IV details OSTRICH2’s key algorithms; Section V discusses the completeness of the algorithms; Section VI shows that OSTRICH2 is extensible with user-defined string functions; Section VII reports evaluation results; and Section VIII concludes with directions for future work.

II. SPECIFICATION LANGUAGE

A. SMT-LIB Standard for Unicode Theory of Strings

We begin with the input language of OSTRICH2. It is based on SMT-LIB 2.6 [11] — in particular, including support for the SMT-LIB Unicode theory and Linear Integer Arithmetic constraints — but additionally also supports some “complex” string functions including transductions. We start by illustrating the supported language features by example and defer the formal description of the grammar to the end of the section.

1) *SMT-LIB Constraints*: Below is a minimal SMT-LIB script to show some core features. The script uses the quantifier-free string theory with linear integer arithmetic (QF_SLIA), contains three string variables `x`, `y`, and `z`, and one integer variable `k`. It asserts constraints on the variables that are explained below. The final statements check the satisfiability of the constraints and produce a satisfying assignment.

```

1 (set-option :produce-models true)
2 (set-logic QF_SLIA)
3
4 (declare-fun x () String)
5 (declare-fun y () String)
6 (declare-fun z () String)
7 (declare-fun k () Int)
8
9 (assert (= (str.len x) (+ k 1)))
10 (assert (= x (str.++ y z)))
11 (assert (str.in_re x (re.+ (re.union (str.to_re "a")
    ↪ (str.to_re "b")))))
12
13 (check-sat)
14 (get-model)

```

Constraints are written in a Lisp-style prefix notation. The first `assert` requires that the length of `x` is equal to `k+1`. The second requires that `x` is equal to the concatenation of `y` and `z`. The third requires that `x` belongs to the regular language $(\{a\} \cup \{b\})^+$, which represents non-empty sequences of `a` and `b` characters.

Standard Boolean connectives are supported. The example below requires `x` to either be the concatenation of `y` and `z` or the concatenation of `z` and `y`.

```

1 (assert (or (= x (str.++ y z)) (= x (str.++ z y))))

```

B. Extensions Beyond the SMT-LIB Standard

OSTRICH2 provides additional features to model string constraints in practical applications. A simple example is the `str.reverse` function. Below, we introduce the transducer, regular expression, and automata extensions.

1) *Transducer-based Operations*: OSTRICH2 supports transducers and prioritised finite-state transducers [30], [31], [29].

A transducer (a.k.a. a *rational transducer*) takes a single string as input and produces a single output string. They enable operations such as HTML encoding (e.g. replacing `&` with `&`) to be defined. Transducers are written as recursive functions with two arguments: the input and the output. The example below shows a `toUpper` transduction. The base case (line 4) applies when both strings are empty.

The recursive case (lines 5–11) asserts three conditions. First, both arguments are non-empty. Second, the head of the output (y) is equal to the upper case version of the head of the input (x). Third, the tails of the two arguments recursively satisfy `toUpper`. To obtain the upper case version of a character, arithmetic is performed on the character codes. In general, a transducer definition can contain multiple mutually recursive functions. The option `:parse-transducers` instructs OSTRICH2 to translate the recursive function definition to an internal transducer representation and apply an automata-based decision procedure to solve the constraints in lines 16–17.

```
1 (set-option :parse-transducers true)
2
3 (define-fun-rec toUpper ((x String) (y String)) Bool
4   (or (and (= x "") (= y ""))
5       (and (not (= x "")) (not (= y ""))
6           (= (char.code (str.head y))
7              (ite (and (<= 97 (char.code (str.head x)))
8                      (<= (char.code (str.head x)) 122))
9                  (- (char.code (str.head x)) 32)
10                  (char.code (str.head x))))
11      (toUpper (str.tail x) (str.tail y))))
12
13 (declare-fun x () String)
14 (declare-fun y () String)
15
16 (assert (= x "Hello World"))
17 (assert (toUpper x y))
```

Prioritised transducers additionally allow certain transitions to take precedence over other transitions. That is, a non-deterministic transduction may only succeed on one path if another cannot succeed. Such transducers are used internally in regular expression handling, described below, but can also be formulated using the `define-fun-rec` notation (not shown here). We discuss these below after we cover extended regular expressions.

2) *Extended Regular Expression Support*: The SMT-LIB standard defines a minimal set of regular expression constructs, all of which are supported by OSTRICH2. In addition, OSTRICH2 can also parse ECMAScript regular expressions, following the 2020 version [32], and supports regular expression features such as look-arounds that cannot be directly expressed in SMT-LIB. A simple usage of ECMAScript regular expressions is shown below, using the `re.from_ecma2020` function. The expression matches either sequences of characters or sequences of digits.

```
1 (assert (str.in_re w (re.from_ecma2020
2   ↳ "[a-z]*|[0-9]*")))
```

To enable full support of ECMAScript character escaping, OSTRICH2 also supports single-quoted strings, in which the normal SMT-LIB character escaping is disabled and strings are passed unmodified to the regular expression parser:

```
1 (assert (str.in_re w (re.from_ecma2020 '\x24[0-9]+')))
```

One notable extension is the use of capture groups and references in `replace` and `extract` functions. The following example shows the replacement of any substring matching `src='.*'` with the text appearing between the single quotation marks. For example `<img src='image.png'>` would become `<img image.png>`. We use the `re.from_ecma2020`

function for convenience; direct functions for regular expression operators—such as `re.capture`—are also available.

```
1 (assert (= x
2   (str.replace_cg_all
3     y
4     (re.from_ecma2020 "src='(.*)'")
5     (_ re.reference 1))))
```

There are several features to note about this example. The first argument of `str.replace_cg_all` is the string to be searched. The second is the regular expression that captures part of the string using the parenthesis notation `(...)`. The final argument is the *replacement pattern* that can be a concatenation of string literals and references to captured text. In this case, only the contents of the first (and only) capture group are used.

Importantly, when matching `src='(.*)'`, precedence rules must be respected. The `*` operator is *greedy* in ECMAScript (and many other regular expression languages), which means that it should match as many characters as possible before the remainder of the match begins. This can be a cause of errors when developers write regular expressions and a challenge for symbolic execution tools using a string constraint solver [33]. For example, if y contained the value `src='a' src='b'`, then there should only be one match of `src='(.*)'` rather than two. In the match, the captured value is `a' src='b'`, that is, the longest string surrounded by `'` symbols. This semantics is respected by OSTRICH2 through the use of *prioritised* transducers [30], [31], [29]. In a prioritised transducer, steps of the transduction can be given priority over the alternatives. When matching against `src='a' src='b'` there is a choice whether the match of `(.*)` should stop at the second `'` or continue. Because `*` is greedy, the match of `(.*)` has priority and matching can only stop if continuing would fail. When replacing the pattern `src='(.*)'` with `src='(.*)?'`, using a *lazy* quantifier `?`, instead the shortest string will be matched.

3) *Automata Representations*: OSTRICH2 also directly supports finite-state automata, which are often more convenient than regular expressions when integrating an SMT solver in applications. This automaton syntax is a bespoke extension specific to OSTRICH2. To the best of our knowledge, no other SMT solver supports this format. The function `re.from_automaton` parses a finite-state automaton given as a string. In the example below the automaton has

- an initial state s_0 ,
- an accepting state s_1 ,
- a transition $s_0 \rightarrow s_1$ `[0, 100]`; (accepting character codes 0–100),
- a loop $s_1 \rightarrow s_1$ `[0, 65535]`; (accepting any UTF-16 code unit).

Then `str.in_re` can be used to test whether the value of an expression belongs to the language of that automaton.

```
1 (assert (str.in_re x
2   (re.from_automaton "automaton value_0 {init s0;
3     ↳ s0 -> s1 [0, 100]; s1 -> s1[0, 65535];
4     ↳ accepting s1;}")))
```

A formal description is given in Appendix A.

4) SMT-LIB Standard for Unicode Theory of Strings:

We begin by outlining the basic syntactic elements for our formulas and the key string and regular expression operations as defined in the SMT-LIB Unicode theory. The fragment below shows the core string and regular expression grammar supported natively by OSTRICH2. Functions not explicitly listed here are internally translated or reduced to equivalent formulas in this grammar. Operators that are extensions beyond SMT-LIB 2.6 are shown underlined.

Formulas:

$$\begin{aligned}
\phi &::= \neg \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \text{Atom} \\
\text{Atom} &::= t_s \sim t_s \mid t_{ar} \sim t_{ar} \mid t_s \in t_{re} \mid \\
&\quad \text{StrPred} \mid \mathcal{T}(t_s, t_s) \\
\text{StrPred} &::= \text{prefixof}(t_s, t_s) \mid \text{suffixof}(t_s, t_s) \mid \\
&\quad \text{contains}(t_s, t_s) \\
t_s &::= \text{BaseStr} \mid \text{StrPos} \mid \text{StrRep} \mid \\
&\quad \text{reverse}(t_s) \\
\text{BaseStr} &::= c_{str} \mid x_{str} \mid \text{concat}(t_s, t_s) \\
\text{StrPos} &::= \text{at}(t_s, t_{ar}) \mid \text{substr}(t_s, t_{ar}, t_{ar}) \\
\text{StrRep} &::= \text{rep}(t_s, t_s, t_s) \mid \text{rep_all}(t_s, t_s, t_s) \mid \\
&\quad \text{rep_re}(t_s, t_{re}, t_s) \mid \text{rep_re_all}(t_s, t_{re}, t_s) \\
t_{ar} &::= c_{int} \mid x_{int} \mid t_{ar} + t_{ar} \mid |t_s| \mid \\
&\quad \text{indexof}(t_s, t_s, t_{ar}) \\
t_{re} &::= \emptyset \mid \Sigma \mid \Sigma^* \mid \text{toRE}(t_s) \mid t_{re} \cdot t_{re} \mid \\
&\quad t_{re} \cup t_{re} \mid t_{re} \cap t_{re} \mid t_{re}^*
\end{aligned}$$

EXPLANATION

- **Formulas (ϕ) and Atoms:** The Boolean formulas ϕ are constructed using the standard logical connectives (negation $\neg$, conjunction $\wedge$, and disjunction $\vee$) along with atomic formulas. The atomic formulas include comparisons on string terms t_s (via a placeholder relation $\sim$), arithmetic terms t_{ar} (also using $\sim$), the membership test $t_s \in t_{re}$, and the dedicated string predicates defined by StrPred . Finally, $\mathcal{T}(t_s, t_s)$ encodes a transducer. The complete grammar for transducers is provided in Appendix B.
- **String Predicates (StrPred):** This subclass of atoms is dedicated to predicates that operate specifically on strings. It includes:
 - $\text{prefixof}(t_s, t_s)$: Checks if the first string is a prefix of the second.
 - $\text{suffixof}(t_s, t_s)$: Checks if the first string is a suffix of the second.
 - $\text{contains}(t_s, t_s)$: Checks if the first string is contained within the second.
- **String Terms (t_s):** The nonterminal t_s is divided into three subclasses and one string function:
 - **BaseStr:** Represents the basic string values. This includes string constants c_s , string variables x_s , or

the concatenation of two string terms using the operator concat .

- **StrPos:** Represents string functions that require an integer parameter. These include:
 - * $\text{at}(t_s, t_{ar})$: Returns the character (as a string) at a specified position.
 - * $\text{substr}(t_s, t_{ar}, t_{ar})$: Returns a substring starting at a given position with a specified length.
- **StrRep:** Contains the string replacement functions, abbreviated here for conciseness. They include:
 - * $\text{rep}(t_s, t_s, t_s)$: Literal replacement.
 - * $\text{rep_all}(t_s, t_s, t_s)$: Replace all occurrences (literal).
 - * $\text{rep_re}(t_s, t_{re}, t_s)$: Regex-based replacement.
 - * $\text{rep_re_all}(t_s, t_{re}, t_s)$: Replace all occurrences based on a regex.

Finally, the operator reverse reverses a string.

- **Arithmetic Terms (t_{ar}):** The arithmetic expressions include integer constants c_{int} , integer variables x_{int} , the sum of two arithmetic expressions $t_{ar} + t_{ar}$, the length function $|t_s|$ (which returns the length of a string), and the function $\text{indexof}(t_s, t_s, t_{ar})$, which returns the index of one string within another as an integer.
- **Regular Expressions (t_{re}):** Regular expressions are defined in a manner consistent with standard mathematical notation:
 - $\emptyset$ denotes the empty language.
 - Σ is the alphabet, and Σ^* represents its Kleene closure.
 - $\text{toRE}(t_s)$ converts a string term into its corresponding regular expression.
 - The operators $\cdot$ (concatenation), $\cup$ (union), $\cap$ (intersection), and $*$ (Kleene star) are used to build more complex regular expressions.

III. SYSTEM ARCHITECTURE

The overall architecture of OSTRICH2 is depicted schematically in Figure 1. OSTRICH2 runs the three solvers (ADT-Str, RCP, and CE-Str) sequentially in a time-sliced configuration. Each solver receives the same input problem and runs independently for a fixed share of the total timeout (e.g. 20 seconds each for a 60-second limit). To reduce repeated work, OSTRICH2 performs common subexpression sharing for selected functions while representing all other expressions separately in tree form. Once started, no further information is shared between solvers as there is currently no established methodology in the literature for aggressive or global sharing of string expressions.

At its core, OSTRICH2 builds on the SMT solver Princess [34], which provides the logical reasoning framework and support for theories such as linear integer arithmetic. OSTRICH2 provides pre-processing and support for the quantifier-free string theories QF_S (pure strings) and QF_SLIA (strings with linear integer arithmetic). Princess

also provides support for algebraic data-types (ADTs), on top of which the ADT-Str string solver is implemented.

The second and third solver are the Regular Constraint Propagation (RCP) and Cost-Enriched String (CE-Str) engine, respectively. These share similar architectures. They begin with a shared string preprocessor, which processes the SMT-LIB input and forwards it to the SMT Core of Princess. Details of the string preprocessing are given in the next section. The SMT Core coordinates with the LIA Solver for linear integer arithmetic and with a solver engine for string-specific reasoning.

Both solvers use automata- and transducer-based representations of sets of strings and supported string functions. A main loop applies proof rules like Automata Intersection, Forward (FWD) and Backward (BWD) Propagation, Nielsen Splitting, and other string-related inference rules. Details of the rules are given in the next section. A string database and an automata database are used for efficiently storing and retrieving string and automata data. The automata database is based on the BRICS Automata Library [35], which provides efficient automaton operations.

For the purposes of presentation, we abstract the constraints received from the SMT core into the following *normal form*:

$$\begin{aligned} S &::= A \mid \neg A \mid S \wedge S \\ A &::= f(x_1, \dots, x_n) \mid x = g(x_1, \dots, x_n) \mid \\ &\quad x \in L \mid x = \sum_i d_i x_i . \end{aligned}$$

In this notation, $x, x_1, \dots, x_n$ are variables of type string or integer, and each d_i is an integer constant. The symbol f denotes a string predicate such as `prefixof`(x_1, x_2) or `suffixof`(x_1, x_2). The symbol g denotes a string function that takes $x_1, \dots, x_n$ as inputs and produces the string x as output, for example $x = \text{concat}(x_1, x_2)$. The notation $x \in L$ expresses that the value of x belongs to a regular language L , which is typically given by a regular expression in SMT-LIB syntax. Finally, the form $x = \sum_i d_i x_i$ represents a linear integer constraint, most commonly arising from length constraints, where the x_i are integer variables and the d_i are constant coefficients. We sometimes write $|t|$ to denote the length of the word represented by the term t . For example, the constraint

```
1 (assert (and (= (str.++ x y) (str.++ y x))
2              (str.in_re x (re.from_ecma2020 "a*ba*"))
3              (str.in_re y (re.from_ecma2020 "a*ca*"))))
```

has the following normal form using an additional variable z

$$z = \text{concat}(x, y) \wedge z = \text{concat}(y, x) \wedge x \in a^*ba^* \wedge y \in a^*ca^* .$$

In the sections below, we give an overview of the three main solver engines. These are: ADT-Str (algebraic data-types), RCP (regular constraint propagation), and CE-Str (cost-enriched strings).

A. ADT-Str: List-Based Solver

The ADT-Str solver builds on the decision procedure for algebraic data-types (ADTs) with size catamorphism implemented in Princess [36]. Algebraic data-types are used to

represent strings using the standard encoding of lists with `nil` and `cons` constructors. The length of a string is computed using the built-in `size` function provided by the ADT solver, mapping every constructor term to the number of constructor occurrences. Other SMT-LIB functions on strings, for instance substring, concatenation, etc., are in ADT-Str encoded using uninterpreted functions and axioms capturing the recursive definition of the string functions. Regular expression matching is implemented using Brzozowski derivatives [37].

ADT-Str is useful, in particular, for computing solutions of string constraints that are outside of the fragments for which the other solvers are complete. ADT-Str can easily handle certain functions that are hard for our propagation algorithms. Those functions include, among others, string-to-integer conversion, and functions like substring and `indexof` that calculate with integer offsets.

B. RCP: Regular Constraint Propagation

Regular Constraint Propagation (RCP) is the newest algorithm that has been implemented in OSTRICH2, based on a subset of proof rules in our paper [29]. The main goal of RCP is to *prove unsatisfiability of the input constraint*. The algorithm handles string functions like concatenation, replace, replaceall, and regular constraints. Other string functions (e.g. reverse, one-way and two-way transducers) are also permitted. These functions permit either exact pre-image or exact post-image computation of regular constraints or both. In general, these images need not be exact, but must at least overapproximate the true pre/post image.

The main idea behind RCP is to propagate regular constraints of the form $x \in L$ to other string variables through forward and backward propagations using the RCP inference rule described in the next section.

ECMAScript Regular Expressions: All three solvers in OSTRICH2 support the SMT-LIB regular expression operators. Full support for ECMAScript regular expression features, including look-arounds, capture groups, and greedy/lazy quantifiers, is implemented in the RCP solver, following the approach in earlier work [29], [38]. Back-references are supported only in the replacement string of `replace` and `replace_all`, not in the matching expression. This implementation uses prioritised transducers to preserve ECMAScript semantics and an intermediate translation to alternating two-way automata for look-arounds. We refer to [29], [38] for full technical details.

C. CE-Str: Cost-Enriched String Engine

The main algorithm of CE-Str (based on [24]) applies backward regular constraint propagation. It uses cost-enriched finite automata (CEFAs) instead of standard finite automata to represent sets of words. These automata are able to capture numerical information about accepted strings, such as the string length, the number of characters appearing before a transition is fired, and so on. This allows length constraints on strings and functions like `indexof` to be directly supported.

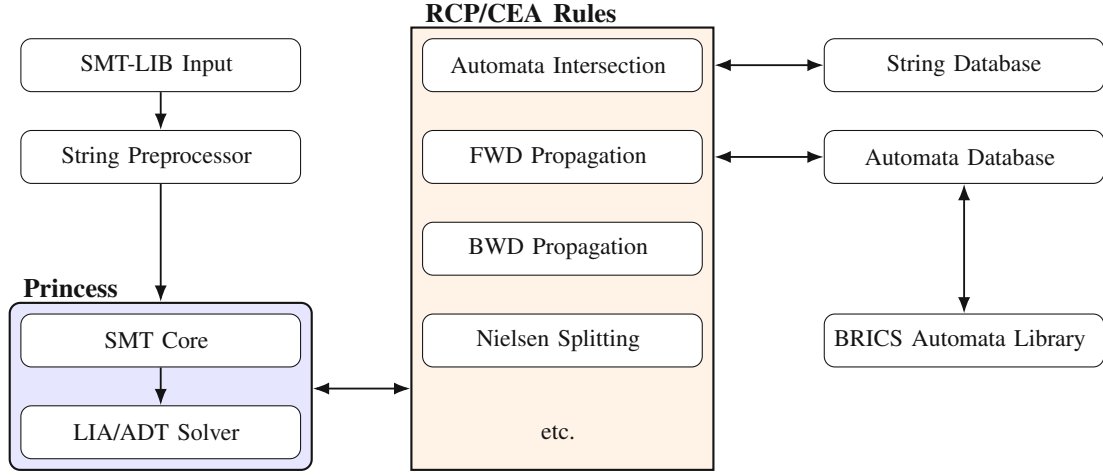


Figure 1. Overall architecture of OSTRICH2: the SMT-LIB input is handled by our string preprocessor and Princess SMT core (with LIA and ADT-Str), while the RCP and CE-Str solvers consist of inference rules that are repeatedly applied using string and automata databases.

In addition, CE-Str can handle counting operators more efficiently by leveraging the numerical information encoded in CEFA. For instance, the regex $a^{\{1,1000\}}$, which accepts strings consisting of 1 to 1000 repetitions of the character a , can be represented as a CEFA with just one state and one transition—compared to the 1000 states and 1000 transitions required by a standard finite automaton.

CE-Str supports a wide range of string functions, including concatenation, replace, replaceall, substring, indexof and length. It provides completeness guarantees for the straight-line fragment of these functions. CE-Str complements RCP and ADT-Str. It performs well on string constraints involving integer type but struggles when the constraints are outside its decidable fragment.

IV. STRING THEORY ALGORITHMS

In this section we describe the core inference rules and algorithmic components that OSTRICH2 applies to solve string constraints. We first present the preprocessing steps shared by the RCP and CE-Str solvers. We then present the *inference rules* applied in the main loops of RCP and CE-Str.

A. Preprocessing

Before applying the core inference rules, OSTRICH2 performs a series of preprocessing steps to enrich the constraint set with auxiliary information and simplify trivial cases. The simplification rules are given below. First we describe the simplification rules used by both RCP and CE-Str, then we describe the additional rules used by RCP only.

1) Common Preprocessing:

Prefix/Contains/Suffix Simplification: We apply the following rewrites when one argument is concrete:

- $\text{prefixof}(s, t)$ / $\text{suffixof}(s, t)$ with concrete s : replace by

$$t \in L(s.*) \quad / \quad t \in L(. * s).$$

- $\text{contains}(t, s)$ with concrete s : replace by

$$t \in L(. * s.*).$$

In *positive* contexts we instead encode:

$$\begin{aligned} \text{prefixof}(s, t) &\rightsquigarrow t = \text{concat}(s, u) \\ \text{suffixof}(s, t) &\rightsquigarrow t = \text{concat}(u, s) \end{aligned}$$

for fresh u . Finally, $\text{prefixof}(t, t)$, $\text{contains}(t, t)$, $\text{suffixof}(t, t)$ are *trivially true* and replaced by the Boolean constant true.

Common Sub-Expression Elimination: While OSTRICH2 does not apply aggressive sharing of terms and expressions, it is useful to merge repeated occurrences of certain operators that are relatively expensive to handle, including expressions $\text{indexof}(x, y, i)$. For expressions e of this kind that occur multiple times in a formula, OSTRICH2 introduces a fresh variable k and adds the equation $k = e$. All occurrences of e in the formula are then rewritten as k . This rewriting ensures that the solver only sees one copy of each complex sub-expression and speeds up subsequent reasoning.

2) *RCP Preprocessing:* The next simplifications are currently used by RCP only.

a) *Length and Character-Count Approximation:* Infers approximate bounds on string lengths and character counts for complex string functions.

- *Derived length constraints:* derive length constraints and character count approximations from word equations and other string functions. E.g. $|x| = |y| + |z|$ is added for $x = \text{concat}(y, z)$.
- *Automaton-transition analysis:* inspect each transition in an automaton A and, for any character a not appearing on any transition, add the char count constraint $|x_a| = 0$ for every x constrained by A .
- *Index-of range constraints:* for each occurrence of $\text{indexof}(x, y, i)$ (which returns the first index of y in x after position i), add implied constraints on the result. E.g., $-1 \leq \text{indexof}(x, y, i) \leq |x|$ captures that the result must be either be -1 or in the interval $[0, |x|]$.

b) *Index-of/Substr/At Rewriting*: Translate `substr`, `at`, and `indexof` into equivalent combinations of string concatenation, length constraints, and regex-membership tests by introducing fresh variables and constraints to encode the positional semantics.

We illustrate one such rewrite for $r = \text{substr}(s, i, n)$, which asserts that r contains the longest contiguous substring of s of length at most n starting at position i . It can be split into three cases.

First we introduce fresh string variables p, r, q with

$$s = \text{concat}(p, r, q) \wedge |p| = i,$$

and then distinguish:

- If $i < 0$, $n \leq 0$, or $i \geq |s|$, then $r = \varepsilon$.
- If $0 \leq i < |s|$ and $i + n \leq |s|$, then $|r| = n$, so r is exactly the length- n slice of s starting at position i .
- If $0 \leq i < |s|$ but $i + n > |s|$, then $|r| = |s| - i$, so r is the substring from position i to the end of s .

B. Inprocessing Rules

There is also a set of lightweight simplification rules that are applied to expressions during proving. Such rules are applied in the local context of a proof goal and are often able to significantly simplify expressions, for instance by evaluating function applications with known arguments or discovering obvious contractions. During inprocessing, equations and assignments of values to variables are propagated to other constraints. There are also certain preprocessing rules that are applied again during inprocessing; for instance, $\neg \text{suffixof}(s, t)$ can be turned into a regular expression constraint as soon as one of the arguments s, t is a known string, but has to be kept unchanged before that.

C. Inference Rules

The main loops of the RCP and CE-Str repeatedly apply proof rules until a branch is closed or a model is found. We discuss the principal rules in this section. The additional rules omitted here that which cover both string-specific reasoning and SMT-core inferences are described in previous work [29], and together with the rules below constitute a complete proof system.

The RCP algorithm assigns priorities to each possible rule application based on an estimation of the workload (e.g. the size of the involved automata) and selects the rule application with the highest priority first. Fairness is ensured by penalizing newer rule applications, and therefore preferring rule applications that have resided in the waiting queue the longest. Priorities are computed as the weighted sum of several criteria:

- *Concrete-argument*: any rule whose input or output language is a ground string is given high priority.
- *Information-gain penalty (backward only)*: rules whose input automata are universal (i.e. accept all words, e.g. $x \in \Sigma^*$) yield little new information in backward propagation and given low priority.

- *Exactness adjustment (forward only)*: forward rules for functions without an exact post-image (e.g. `replaceall` with symbolic patterns) are given low priority.
- *Cost-based penalty*: a weight proportional to the combined size of the input and result automata is subtracted.

Differences between the two solvers in their application of the rules is included in the descriptions. The proof rules may cause branching in the proof search. If a satisfying assignment is found on a branch, it witnesses that the constraint is satisfiable. The Close rule detects unsatisfiable constraints. If all branches are closed, the constraint is unsatisfiable.

a) *Regex-to-Automata*: OSTRICH2 constructs an automaton A for each regular membership constraint in $x \in L$. The automaton is stored in the automaton database, which detects equivalent automata to avoid duplication. In the remaining rules, we make the automaton explicit by writing $x \in L(A)$. The CE-Str solver creates cost-enriched automata. Initially these automata do not track any costs. Cost tracking is introduced during the RCP rule below.

b) *BreakCyclicEquations*: OSTRICH2 detects strongly connected components in the variable-dependency graph induced by concatenation equations of the form

$$x = \text{concat}(y, z) \quad \text{and} \quad y = \text{concat}(a, x).$$

For each cycle it removes one equation and, for every remaining equation $v = \text{concat}(u, w)$ in that cycle, adds a trivial emptiness constraint on one argument (e.g. $w = \varepsilon$ or $u = \varepsilon$) to break the dependency.

For example, from

$$x = \text{concat}(y, z) \quad \wedge \quad y = \text{concat}(a, x)$$

we might remove the first equation and introduce

$$y = \text{concat}(a, x), \quad z = \varepsilon, \quad a = \varepsilon,$$

yielding an acyclic set of equations. This transformation is *sound*, because in any finite model of the original cyclic equations at least one concatenation argument must be empty, and *complete*, since any solution of the resulting acyclic system (with the added emptiness constraints) automatically satisfies the dropped equations.

c) *Equation Decomposition*: OSTRICH2 uses two simple heuristics to simplify word equations. In what follows, z, x_1, x_2, y_1, y_2 are arbitrary string terms (each may be a variable or a constant). First, when one side is a constant string w and the other is a concatenation

$$w = \text{concat}(x_1, x_2),$$

in which the length of x_1 is known, OSTRICH2 matches the first $|x_1|$ characters of w with x_1 and the remaining characters with x_2 . Second, if the same variable appears in two concatenations,

$$z = \text{concat}(x_1, x_2) \quad \text{and} \quad z = \text{concat}(y_1, y_2),$$

then whenever $|x_1| = |y_1|$ we infer $x_1 = y_1$ and $x_2 = y_2$. These heuristics often resolve equations without full case splitting.

d) *Close*: If for some variable x we have constraints

$$x \in L(A_1) \wedge \dots \wedge x \in L(A_k)$$

and $\bigcap_{i=1}^k L(A_i) = \emptyset$, OSTRICH2 derives a contradiction and close the branch.

e) *Intersection*: When the `+eager` flag is on, OSTRICH2 maintains at most one automaton per variable by replacing $x \in L(A) \wedge x \in L(B)$ with $x \in L(A \cap B)$.

f) *LengthAbstraction*: From length inequalities (e.g. $|x| \leq |y| + 5$), derive lower/upper bounds on $|x|$, then assert $x \in L(A)$ where A accepts any word within the derived length bounds.

g) *RCP (Regular Constraint Propagation)*: Propagate $x_1 \in L_1, \dots, x_n \in L_n$ forwards through string functions $x = f(x_1, \dots, x_n)$, or $x \in L$ backwards through f .

For example, on $z = \text{concat}(x, y) \wedge z = \text{concat}(y, x) \wedge x \in a^*ba^* \wedge y \in a^*ca^*$ we can propagate $x \in a^*ba^*$ and $y \in a^*ca^*$ forwards (from input to output) through the string concatenation function concat in $z = \text{concat}(x, y)$ to obtain $z \in a^*ba^*a^*ca^*$. Similarly, we can propagate forwards through $z = \text{concat}(y, x)$ and derive $z \in a^*ca^*a^*ba^*$. Since there are no words matching both $a^*ba^*a^*ca^*$ and $a^*ca^*a^*ba^*$ we can conclude—using the Close rule—that the constraint is unsatisfiable.

Backwards propagation may result in branching. For example, if $x = \text{concat}(y, z)$ and $x = ab$, then either $y = ab \wedge z = \varepsilon$ or $y = a \wedge z = b$ or $y = \varepsilon \wedge z = ab$.

When the variable constraints are given by standard finite automata, a rich selection of string functions support exact forwards and backwards propagation. These include `concat`, `reverse`, and `replace` and `replaceall`, where the replacement pattern can include string variables or references to capture groups in the search pattern. In general, any function can be supported using over-approximations of the pre- and post-images, but without completeness guarantees.

The CE-Str solver only applies backwards propagation and represents pre-images using cost-enriched automata. This allows functions such as `indexof` and `length` to be supported (by introducing costs to the automata), but restricts, for example, which versions of `replace` and `replaceall` can be analysed precisely.

h) *NielsenSplitter*: OSTRICH2 invokes Nielsen’s transformation [39], [40] on any pair of equations in our normal form

$$z = \text{concat}(x_1, x_2) \quad \text{and} \quad z = \text{concat}(y_1, y_2),$$

where each of z, x_i, y_i may be a variable or a constant. We introduce a fresh string variable w and split the proof into two *prefix-alignment* cases, guided by current length information in a similar way as was done in Norn [3]. That is

$$|x_1| \geq |y_1|, \quad x_1 = \text{concat}(y_1, w), \quad \text{concat}(w, x_2) = y_2$$

or

$$|y_1| \geq |x_1|, \quad y_1 = \text{concat}(x_1, w), \quad \text{concat}(w, y_2) = x_2.$$

In each branch we align the longer prefix against the shorter one, adding both the corresponding concatenation equalities and the derived length equality (e.g. $|x_1| = |y_1| + |w|$ in the first case).

i) *String-Integer-Conversions*: For handling expressions $\text{str_to_int}(s) = n$ or $\text{int_to_str}(n) = s$, OSTRICH2 systematically explores the possible values of n . As soon as n has a concrete value, inprocessing rules (Section IV-B) apply that replace the function application with an equation or a regular expression constraint describing the possible values of s . To make this exploration perform well in practical cases, the LIA solver utilized in OSTRICH2 applies interval constraint propagation to narrow down the range of possible values of n as much as possible [41]. The interval for n is then subdivided, until eventually only one possible value for n remains. This search will not terminate in general, since the domain of n can be infinite, but it tends to derive solutions or contradictions quickly in many practical cases.

j) *Index Computation*: When handling $\text{indexof}(s, p, k)$ on two concrete strings s, p , OSTRICH2 systematically explores possible values of k , taking into account the fact that any value k not satisfying $0 \leq k \leq |s| - |p|$ will necessarily lead to the result -1 . In the “in-range” branch OSTRICH2 applies interval constraint propagation to determine the possible values of k , sub-dividing the interval of values until a concrete value k has been chosen. As soon as all arguments of `indexof` have concrete values, inprocessing rules (Section IV-B) can evaluate the function.

k) *Cut Rule*: When all other rules have been exhausted, OSTRICH2 applies cuts to introduce a candidate solution for a string variable x . For this, OSTRICH2 collects every regular-language membership constraint $x \in L_i$ together with any length bounds $\ell \leq |x| \leq u$. From the intersection of the L_i (and respecting ℓ, u), OSTRICH2 extracts a single accepted word w via a standard automaton search. We then split on the two exhaustive cases

$$x = w \quad \text{vs.} \quad x \notin \{w\}.$$

This rule is always sound because any satisfying assignment for x either equals the chosen w or lies outside $\{w\}$ and guarantees that each string variable will eventually be grounded to a concrete value (or shown impossible).

V. COMPLETENESS RESULTS

The RCP solver is (in principle) complete for the straight-line fragment [4] and chain-free fragment [42] of string constraints, in the sense that the underlying proof system can show unsatisfiability of unsatisfiable instances from these fragments, when applying the rules in an appropriate (easy) order. However, the usage of priorities means the rules may be applied in a different order, causing the proof to fail. In practice, failures for these fragments are rare.

In the straight-line fragment we require that the constraints are in normal form as in Section IV-C, and variables can be ordered $x_1, \dots, x_n$ such that for x_i there is exactly one equational constraint $x_i = f_i(x_1, \dots, x_{i-1})$, where f_i is a

string function such that the pre-image of a regular set is regular. The definition of the chain-free fragment is a bit involved, in essence it relaxes the assumption on the form of the equation and the number of equations, but still requires that there are no “cycles” of dependencies. For the straight-line fragment, we can propagate backwards and the instance is unsatisfiable if and only if all branches are closed using Close rule. For the chain-free fragment [42] we can adapt the original proof (which works for a different solver) to the rules used in OSTRICH2.

The CE-Str solver also provides a completeness guarantee for a version of the straight-line fragment that includes string constraints and linear integer arithmetic [24]. The supported string functions are those where the pre-image of cost-enriched automata constraints can be expressed using cost-enriched automata. This includes indexof and length as well as some versions of replace and replaceall.

VI. EXTENSIBILITY

An important feature of OSTRICH2 is the possibility to extend the solver with minimal effort. OSTRICH2’s PreOp trait lets users add new string functions by overriding three core methods. This can be done in either Scala (the main language used to implement OSTRICH2) or in Java. Below we show each method in turn, together with the ReversePreOp implementation¹.

First, the method `apply` computes the *pre-image* of a regular language under f , which is the core operation needed for backward propagation in RCP. The method is passed any existing automata constraints on the r arguments (which may be ignored) and the automaton $\mathcal{A}$ representing the result language. The method returns an iterator over tuples of automata whose images under f lie inside $\mathcal{A}$, plus (optionally) the subset of the input constraints actually used.

```
1 override def apply(
2   argCs: Seq[Seq[Automaton]],
3   resultC: Automaton
4 ) : (Iterator[Seq[Automaton]], Seq[Seq[Automaton]]) = {
5   (Iterator(Seq(ReverseAutomaton(resultC))), List())
6 }
```

Second, the method `eval` performs *concrete* evaluation on ground strings: when all arguments are known, it simply returns the result of applying f .

```
1 override def eval(
2   args: Seq[Seq[Int]]
3 ) : Option[Seq[Int]] = Some(args.head.reverse)
```

Third, `forwardApprox` computes a (sound) *post-image* of input languages under f , used for forward propagation. While returning the universal automaton is always correct, a tighter approximation greatly improves performance. For `str.reverse`, one can exactly compute this by intersecting any input automata and then reversing the resulting automaton.

¹This snippet is abbreviated for clarity. The full implementation in `ReversePreOp.scala` includes additional automaton-type matching and error handling.

```
1 override def forwardApprox(
2   argCs: Seq[Seq[Automaton]]
3 ) : Automaton = {
4   val prod = ProductAutomaton(argCs)
5   ReverseAutomaton(prod)
6 }
```

Finally, the new string function has to be registered in the core string theory class of OSTRICH2. In the file `OstrichStringTheory.scala`, we register `str.reverse` by first declaring its SMT-side function symbol:

```
1 val str_reverse = MonoSortedIFunction(
2   "str.reverse",
3   List(StringSort), StringSort, true, false
4 )
```

We then add it to the `extraStringFunctions` list, pairing the name, the `IFunction`, our `ReversePreOp` implementation, and the simple lambdas that pick out the input and output terms:

```
1 val extraStringFunctions = List(
2   ("str.reverse", str_reverse, ReversePreOp, a =>
3     List(a(0)), a => a(1)))
```

With these two lines in place, any occurrence of `str.reverse` in the SMT-LIB script is recognized by OSTRICH2 and automatically dispatched to our `ReversePreOp` for pre-image computation, concrete evaluation, and forward-approximation.

VII. EXPERIMENTS

A. Benchmark suites and experimental setup

We evaluate our solver on a representative selection of SMT-LIB benchmarks, including all instances from the SMT-LIB 2025 benchmark release². From the combined pool of roughly 100 000 QF_S and QF_SLIA problems, we randomly sample 2 000 instances proportional to each track’s share of the total, so that our test set reflects the underlying distribution of problem types. The experiments were conducted on a MacBook Pro with 16 GB of RAM, running macOS Sonoma 14.5. The system was powered by an Apple M3 chip. The timeout for each experiment was set to 60 seconds.

ECMAScript Regex Benchmarks.: Our experimental evaluation in this paper does not include benchmarks making extensive use of ECMAScript-specific regular expression features such as look-arounds or capture groups. A large-scale evaluation of this functionality was presented in [38], using a benchmark set of approximately 8,800 ECMAScript-style regex patterns extracted from real-world web forms. The correctness of the semantics was validated in [29] by comparing OSTRICH’s results against JavaScript’s reference implementation. The ECMAScript regex component in OSTRICH2 is identical to that used in those earlier evaluations. Therefore, we omit repeating those experiments here and focus on broader SMT-LIB benchmarks.

²<https://zenodo.org/communities/smt-lib/>

B. Performance evaluation

The experiments are conducted on the following solvers: *cvc5* 1.2.0 [19], *Z3* 4.15 [12], *Z3-alpha* [13], *Z3-Noodler* 1.3.0 [14], *OSTRICH1* (the original *OSTRICH*) [4], and different engines of *OSTRICH2*³ and *OSTRICH2* running the engines in a time-sliced portfolio using the flag. For the RCP engine we evaluate two complementary configurations: one mode that combines forward and backward propagation without Nielsen splitting (using the flags `+F+B-N`), and another mode (using the flags `-F+B+N`) that disables forward propagation in order to emphasize Nielsen’s equation decomposition. In practice, combining forward propagation and splitting yields limited synergy: the extra word equations generated by Nielsen splitting create significant propagation overhead, so omitting forward propagation often improves overall solving performance.

	<i>cvc5</i>	<i>Z3</i>	<i>Z3-alpha</i>	<i>Z3-Noodler</i>	<i>OSTRICH1</i>
SAT	1155	1097	1123	1201	825
UNSAT	729	728	728	744	664
unknown/timeout	116	175	149	55	511
Solved	1884	1825	1851	1945	1489
Total time [s]	7770	12158	10523	3660	35711

Figure 2. The experiment result of *cvc5*, *Z3*, *Z3-alpha*, *Z3-Noodler* and *OSTRICH1* on the SMT-LIB’25 benchmarks. Total time (in seconds) includes all instances, with each unknown/timeout counted as 60s.

	<i>OSTRICH2</i>	ADT	CE	RCP-F+B+N	RCP-F+B-N
SAT	1147	610	970	1067	1042
UNSAT	758	619	679	747	750
unknown/timeout	95	771	351	186	208
Solved	1905	1229	1649	1814	1792
Total time [s]	14824	53689	24600	17680	19900

Figure 3. The experiment result of different engines of *OSTRICH2* on the SMT-LIB’25 benchmarks. The flags `+F+B-N` enable forwards/backwards propagation and disable the NielsenSplitter. The flags `-F+B+N` disable forwards propagation and enable backwards propagation and the NielsenSplitter. Total time (in seconds) includes all instances, with each unknown/timeout counted as 60s.

Figure 2 shows that among the off-the-shelf solvers, *Z3-Noodler* leads with 1945 solved problems, followed by *cvc5* (1884), *Z3-alpha* (1851), *Z3* (1825), and *OSTRICH1* (1489). Figure 3 shows that both RCP configurations (1814 and 1792 solves) and CE-Str (1649) represent clear upgrades over *OSTRICH1*, whereas the ADT-Str engine (1229) trails—largely due to its simple algorithm. These per-engine gains can be attributed to improved preprocessing, an extended set of proof rules, and, for CE-Str, a different underlying solving technique.

In the full *OSTRICH2* configuration, these upgraded engines are run in a time-sliced portfolio, which not only inherits the improvements of the individual modes but also exploits their complementary strengths, as each engine solves benchmarks that the others miss. This complementarity lifts

OSTRICH2’s overall performance close to the top solvers in the field.

When we combine all four engines in a time-sliced portfolio (15 seconds for each mode for a 60-second limit), *OSTRICH2* solves 1905 problems, putting it just behind *Z3-Noodler* and ahead of the rest of the field. The portfolio not only inherits the individual gains of each engine but also benefits from their complementary strengths, as each solves benchmarks the others miss. We further observe that *Z3-Noodler* excels on the SAT instances, while *OSTRICH2* performs particularly well on the UNSAT cases.

Finally, although *Z3-Noodler* does not support `replace_all`, only 2.4% of SMT-LIB instances use this operator, so its absence has only a modest effect on overall rankings. Figure 4 shows the results on the benchmark set restricted to instances without `replace_all`. Among the 35 `replace_all` benchmarks in our SMT-LIB sample, *OSTRICH2* solves 25, while *cvc5*, *Z3*, and *Z3-alpha* each solve only two instances, and *Z3-Noodler* solves none. This shows that most solvers provide limited support for this operator. When we exclude the `replace_all` benchmarks, the rankings remain broadly unchanged; *OSTRICH2* drops slightly below *cvc5* in relative performance.

We remark that, since our evaluation uses a stratified random sample of 2000 instances drawn from the full pool of roughly 100000 `QF_S` and `QF_SLIA` benchmarks, there is inevitably some statistical variance in the exact per-solver ranking; nonetheless, the fact that *OSTRICH2* solves almost as many problems as the leader demonstrates its overall competitiveness.

	<i>cvc5</i>	<i>Z3</i>	<i>Z3-alpha</i>	<i>Z3-Noodler</i>	<i>OSTRICH2</i>
SAT	1155	1097	1123	1201	1141
UNSAT	727	726	726	744	739
unknown/timeout	83	142	116	20	85
Solved	1882	1823	1849	1945	1880
Total time [s]	5790	10174	8536	1617	14070

Figure 4. The experiment result of *cvc5*, *Z3*, *Z3-alpha*, *Z3-Noodler* and *OSTRICH2* on the SMT-LIB’25 benchmarks **excluding** `replace_all`. Total time (in seconds) includes all instances, with each unknown/timeout counted as 60s.

VIII. CONCLUSION

This paper has strived to provide a concise introduction of the string solver *OSTRICH2*, which inputs constraints in an extension of the SMT-LIB theory of Unicode Strings with transducers (through mutual recursion) and ECMAScript regular expressions, among others. We have also empirically demonstrated its competitiveness with other string solvers on SMT-COMP benchmarks, particularly on unsatisfiable instances. Interested readers, who are interested in participating in the development and/or applications of *OSTRICH2*, are wholeheartedly encouraged to get in touch with us.

³<https://doi.org/10.5281/zenodo.15378521>

ACKNOWLEDGEMENTS

We thank EPSRC [EP/T00021X/1], European Research Council (LASD, 101089343), and the Swedish Research Council (grant 2021-06327) for their support.

REFERENCES

- [1] C. Kern, “Securing the tangled web,” *Commun. ACM*, vol. 57, no. 9, pp. 38–47, Sep. 2014.
- [2] A. W. Lin and P. Barceló, “String solving with word equations and transducers: towards a logic for analysing mutation XSS,” in *POPL*, ACM, 2016, pp. 123–136.
- [3] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezzine, P. Rümmer, and J. Stenman, “Norn: An SMT solver for string constraints,” in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 462–469. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_29
- [4] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, “Decision procedures for path feasibility of string-manipulating programs with complex operations,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.
- [5] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *S&P*, 2010, pp. 513–528.
- [6] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, “HAMPI: a solver for string constraints,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, G. Rothermel and L. K. Dillon, Eds. ACM, 2009, pp. 105–116. [Online]. Available: <https://doi.org/10.1145/1572272.1572286>
- [7] N. Bjørner, N. Tillmann, and A. Voronkov, “Path feasibility analysis for string-manipulating programs,” in *TACAS*, 2009, pp. 307–321.
- [8] R. Amadini, “A survey on string constraint solving,” *ACM Comput. Surv.*, vol. 55, no. 2, pp. 16:1–16:38, 2023. [Online]. Available: <https://doi.org/10.1145/3484198>
- [9] N. Runfta, “A billion SMT queries a day (invited paper),” in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13371. Springer, 2022, pp. 3–18. [Online]. Available: https://doi.org/10.1007/978-3-031-13185-1_1
- [10] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Runfta, O. Tkachuk, and C. Varming, “Semantic-based automated reasoning for AWS access policies using SMT,” in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. S. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–9. [Online]. Available: <https://doi.org/10.23919/FMCAD.2018.8602994>
- [11] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” www.smt-lib.org, 2016.
- [12] L. de Moura and N. Bjørner, “Z3: an efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963, 04 2008, pp. 337–340.
- [13] Z. Lu, S. Siemer, P. Jha, J. Day, F. Manea, and V. Ganesh, “Layered and staged monte carlo tree search for smt strategy synthesis,” in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, K. Larson, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2024, pp. 1907–1915, main Track. [Online]. Available: <https://doi.org/10.24963/ijcai.2024/211>
- [14] F. Blahoudek, Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Sif, “Word equations in synergy with regular constraints,” in *Formal Methods, M. Chechik, J.-P. Katoen, and M. Leucker*, Eds. Cham: Springer International Publishing, 2023, pp. 403–423.
- [15] V. Havlena, L. Holík, O. Lengál, and J. Sif, “Cooking String-Integer Conversions with Noodles,” in *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Chakraborty and J.-H. R. Jiang, Eds., vol. 305. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 14:1–14:19. [Online]. Available: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2024.14>
- [16] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Sif, “Solving string constraints with lengths by stabilization,” *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: <https://doi.org/10.1145/3622872>
- [17] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Sif, “Z3-noodler: An automata-based string solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 24–33.
- [18] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh, “An smt solver for regular expressions and linear arithmetic over string length,” in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 289–312.
- [19] H. Barbosa, C. 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*, 01 2022, pp. 415–442.
- [20] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh, “Z3str4: A multi-armed string solver,” in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 389–406. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-6_21
- [21] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezzine, and P. Rümmer, “Flatten and conquer: a framework for efficient analysis of string constraints,” *SIGPLAN Not.*, vol. 52, no. 6, p. 602–617, Jun. 2017. [Online]. Available: <https://doi.org/10.1145/3140587.3062384>
- [22] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezzine, and P. Ruemmer, “Trau: Smt solver for string constraints,” *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–5, 2018. [Online]. Available: <https://api.semanticscholar.org/CorpusID:53962814>
- [23] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. Diep, L. Holík, D. Hu, W.-L. Tsai, Z. Wu, and D.-D. Yen, “Solving not-substring constraint with flat abstraction,” in *Programming Languages and Systems*, H. Oh, Ed. Cham: Springer International Publishing, 2021, pp. 305–320.
- [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] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.
- [26] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An automata-based string analysis tool for php,” 05 2010, pp. 154–157.
- [27] M.-T. Trinh, D.-H. Chu, and J. Jaffar, “S3: A symbolic string solver for vulnerability detection in web applications,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1232–1243. [Online]. Available: <https://doi.org/10.1145/2660267.2660372>
- [28] M. Berzish, V. Ganesh, and Y. Zheng, “Z3str3: A string solver with theory-aware heuristics,” in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 55–59.
- [29] 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>
- [30] M. Berglund, F. Drewes, and B. van der Merwe, “Analyzing catastrophic backtracking behavior in practical regular expression matching,” in *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014*, ser. EPTCS, Z. Ésik and Z. Fülöp, Eds., vol. 151, 2014, pp. 109–123. [Online]. Available: <https://doi.org/10.4204/EPTCS.151.7>
- [31] M. Berglund and B. van der Merwe, “On the semantics of regular expression parsing in the wild,” *Theoretical Computer Science*, vol. 679, pp. 69 – 82, 2017.
- [32] J. Harband and K. Smith, “ECMAScript 2020 language specification, 11th edition,” 2020, <https://262.ecma-international.org/11.0/>.

- [33] 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.*, 2019, pp. 425–438. [Online]. Available: <https://doi.org/10.1145/3314221.3314645>
- [34] P. Rümmer, “A constraint sequent calculus for first-order logic with linear integer arithmetic,” in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2008, pp. 274–289.
- [35] A. Möller, “dk.brics.automaton – finite-state automata and regular expressions for java,” <https://www.brics.dk/automaton>, 2021, accessed: 2022-06-23.
- [36] H. Hojjat and P. Rümmer, “Deciding and interpolating algebraic data types by reduction,” in *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, T. Jebelean, V. Negru, D. Petcu, D. Zaharie, T. Ida, and S. M. Watt, Eds. IEEE Computer Society, 2017, pp. 145–152. [Online]. Available: <https://doi.org/10.1109/SYNASC.2017.00033>
- [37] J. A. Brzozowski, “Derivatives of regular expressions,” *J. ACM*, vol. 11, no. 4, p. 481–494, oct 1964. [Online]. Available: <https://doi.org/10.1145/321239.321249>
- [38] B. Eriksson, A. Stjerna, R. De 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*, ser. CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 549–563. [Online]. Available: <https://doi.org/10.1145/3576915.3616582>
- [39] A. Lentin, *Equations dans les Monoides Libres*. Gauthier-Villars, Paris, 1972.
- [40] V. Diekert, “Makanin’s Algorithm,” in *Algebraic Combinatorics on Words*, ser. Encyclopedia of Mathematics and its Applications, M. Lothaire, Ed. Cambridge University Press, 2002, vol. 90, ch. 12, pp. 387–442.
- [41] P. Backeman, P. Rümmer, and A. Zeljic, “Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic,” *Formal Methods Syst. Des.*, vol. 57, no. 2, pp. 121–156, 2021. [Online]. Available: <https://doi.org/10.1007/s10703-021-00372-6>
- [42] P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janků, “Chain-free string constraints,” in *Automated Technology for Verification and Analysis*, Y.-F. Chen, C.-H. Cheng, and J. Esparza, Eds. Cham: Springer International Publishing, 2019, pp. 277–293.

APPENDIX A AUTOMATA DEFINITION

```

Automaton ::= automaton Ident { init State ;
                               Tr* accepting State (, State)* ; }
Tr          ::= State -> State [ Int , Int ] ;
State       ::= Ident
Ident       ::= [A-Za-z0-9_]+
Int         ::= [0-9]+

```

Ranges are inclusive and denote Unicode code points as accepted by the current `re.from_automaton` parser (0–65535)⁴. States referenced in transitions are implicitly declared. Whitespace is insignificant.

APPENDIX B TRANSDUCER DEFINITIONS

OSTRICH encodes *transducers* as a collection of mutually recursive Boolean functions in SMT-LIB, where each function corresponds to a distinct state in the transducer. A transducer consists of a set of such states that, at each step, inspect the first character(s) of the input and output strings, select a transition based on guarded conditions, and then recurse on the remaining substrings until termination.

Formally, the behaviour of each state function is structured as follows:

- 1) Inspect the leading characters of the input and output strings using `str.head`.
- 2) Evaluate guard conditions to determine the appropriate transition.
- 3) Invoke another state function (possibly the same one) on the residual substrings obtained via `str.tail`.

Acceptance is defined via *base cases*, typically when both input and output have been fully consumed, expressed as `(= x "")` and `(= y "")`. Character comparisons are performed using Unicode code points, with numeric ranges used to express character classes (e.g., lowercase letters correspond to codes 97–122).

A. Example 1: General Template

In the template below, the transducer is defined by two mutually recursive state functions, `S` and `S2`, each taking as arguments the remaining portions of the input (`x`) and output (`y`) strings. The first clause of each state specifies the *base case*, here accepting when both `x` and `y` are empty. Subsequent clauses correspond to guarded transitions: in `S`, the first transition matches and deletes an ‘a’ from the input (consuming only `x` and leaving `y` unchanged), while the second transition copies the current character from input to output (consuming one symbol from each). Each transition invokes the appropriate successor state on the residual strings obtained via `str.tail`. The second state, `S2`, is structurally similar, here illustrated with only a copying transition. This

⁴While OSTRICH2 supports reasoning over the full Unicode range, the automaton parser is presently limited to BMP code points.

structure generalises to arbitrary transducers by adding states, refining guard conditions, and controlling which side(s) of the input-output pair are consumed in each branch.

```

1 (set-option :parse-transducers true)
2 ; One state per function, (x,y) are the remaining
  ⇨ input/output.
3 (define-funs-rec
4   ((S ((x String) (y String)) Bool)
5    (S2 ((x String) (y String)) Bool))
6   (
7     ; S: base case = accept when both consumed
8     (or (and (= x "") (= y ""))
9         ; transition 1: delete 'a' from input
10        (and (not (= x ""))
11              (= (str.head x) (char.from-int 97)) ; 'a'
12              (S (str.tail x) y))
13        ; transition 2: copy char (consume both)
14        (and (not (= x "")) (not (= y ""))
15              (= (str.head x) (str.head y))
16              (S (str.tail x) (str.tail y))))
17        ; S2: another state if needed...
18        (or (and (= x "") (= y ""))
19              (and (not (= x "")) (not (= y ""))
20                    (= (str.head x) (str.head y))
21                    (S2 (str.tail x) (str.tail y))))
22    )

```

B. Example 2: toUpper

The `toUpper` transducer illustrates a length-preserving character transformation. It consists of a single state, `toUpper`, which accepts when both input `x` and output `y` are empty. Otherwise, the guard requires that both strings be non-empty, and the head character of the output is constrained to be either the uppercase equivalent of the head of the input (if the input character is a lowercase letter, identified by Unicode codes 97–122) or an exact copy of the input character in all other cases. This transformation is expressed using the `ite` construct, subtracting 32 from the character code to obtain the uppercase form when applicable. The function then recurses on the tails of both strings, ensuring that the transformation is applied position-wise until the entire input has been processed.

```

1 (set-option :parse-transducers true)
2
3 (define-fun-rec toUpper ((x String) (y String)) Bool
4   (or (and (= x "") (= y ""))
5       (and (not (= x "")) (not (= y ""))
6           (= (char.code (str.head y))
7              (ite (and (<= 97 (char.code (str.head x)))
8                      (<= (char.code (str.head x)) 122))
9                  (- (char.code (str.head x)) 32)
10                  (char.code (str.head x))))
11         (toUpper (str.tail x) (str.tail y))))

```

C. Example 3: extract1st

The `extract1st` transducer demonstrates a non-length-preserving transformation involving multiple states. Its purpose is to scan the input `x` for the first occurrence of the character `'='` (Unicode code 61), copy the subsequent characters into the output `y` until the next `'='` is encountered, and then skip the remainder of the input. The initial state, `extract1st`, advances through the input without producing output until it finds the first `'='`, at which point it transitions to `extract1st_2`. In this second state, the transducer consumes characters from both input and output in lockstep, copying them directly unless another `'='` is reached, which

triggers a transition to `extract1st_3`. The final state, `extract1st_3`, consumes the remaining input without producing further output, thereby terminating the extraction. For example, given the input string `x = "foo=bar"`, the transducer outputs `y = "bar"`.

```

1 (set-option :parse-transducers true)
2
3 (define-funs-rec ((extract1st ((x String) (y String))
  ⇨ Bool)
4   (extract1st_2 ((x String) (y String)) Bool)
5   (extract1st_3 ((x String) (y String)) Bool)) (
6     ;
7     ; extract1st
8     (or (and (= x "") (= y ""))
9         (and (not (= x ""))
10              (not (= (str.head x) (char.from-int 61))) ; not
11                  ⇨ '='
12              (extract1st (str.tail x) y))
13              (and (not (= x ""))
14                    (= (str.head x) (char.from-int 61)) ; '='
15                    (extract1st_2 (str.tail x) y)))
16     ;
17     ; extract1st_2
18     (or (and (= x "") (= y ""))
19         (and (not (= x "")) (not (= y ""))
20              (= (str.head x) (str.head y))
21              (not (= (str.head x) (char.from-int 61))) ; not
22                  ⇨ '='
23              (extract1st_2 (str.tail x) (str.tail y)))
24              (and (not (= x ""))
25                    (= (str.head x) (char.from-int 61)) ; '='
26                    (extract1st_3 (str.tail x) y)))
27     ;
28     ; extract1st_3
29     (or (and (= x "") (= y ""))
30         (and (not (= x ""))
31              (extract1st_3 (str.tail x) y)))
32   )

```

D. Grammar for OSTRICH transducers

The following grammar summarises the fragment of SMT-LIB used by OSTRICH for defining such transducers.

```

Transducer ::= (define-fun-rec StateSig Body)
             | (define-funs-rec (StateSig+) (Body+))

StateSig  ::= (Id ((x String) (y String)) Bool)

Body      ::= (or Clause+)

Clause    ::= BaseCase | Transition

BaseCase  ::= (and Guard*)

Transition ::= (and Guard+ Call)

Call      ::= (Id XArg YArg)

XArg      ::= x | (str.tail x)

YArg      ::= y | (str.tail y)

Guard     ::= (= Term Term)
             | (not Guard)
             | (and Guard+)
             | (or Guard+)
             | (<= Int Int)

Term      ::= Char | (str.head x) | (str.head y)
             | (char.code Term)
             | (ite Guard Term Term)

Char      ::= (char.from-int Int)
             | (str.head x)
             | (str.head y)

Int       ::= Numeral | (char.code Term)
             | (+ Int Int)
             | (- Int Int)

```

In this grammar, each `StateSig` corresponds to one *state* in the transducer. The `Body` of a state consists of one or more `Clauses` combined with `or`, where each clause represents either:

- a *base case*, such as `(and (= x "") (= y ""))`,
or
- a *transition* guarded by conditions on the heads of `x` and `y` and leading to a recursive `Call` on their tails.

The `Guard` syntax captures conditions such as character equality, inequality, or membership in a Unicode range (as in `toUpper`, which checks lowercase via `(<= 97 (char.code (str.head x)))` and `(<= (char.code (str.head x)) 122)`). `XArg` and `YArg` indicate whether a transition consumes from the input, output, or both. This is the key distinction between length-preserving transformations (both consumed) and non-length-preserving ones (only one consumed at a time).

A Formal Y86 Simulator with CHERI Features

Carl Kwan , Yutong Xin, William D. Young
The University of Texas at Austin, Austin, TX, USA
{carlkwan,maxxin,byoung}@cs.utexas.edu



Abstract—We present a formal executable model of CHERI architectural features integrated with a formalized Y86 ISA. CHERI is an extension of conventional hardware ISAs centralized around capabilities, which are descriptions of permissions at the hardware level that can be used in place of pointers. CHERI enables fine-grained memory protection and highly scalable software compartmentalization to mitigate security vulnerabilities beyond what can be done by current architectures. We use our formal model to prove the memory protection and security features of CHERI itself, prototype extensions to CHERI, and verify the correctness of machine code involving capabilities. Since it is executable, our model also serves as a *symbolic simulator* of a CHERI augmented x86-like processor, allowing step-by-step validation of CHERI capability features in a controlled, formal environment.

We are motivated by the adoption of CHERI by industry leaders that design real-world hardware, the need for industrial-strength tools to perform formal CHERI analyses, and the current development of CHERI extensions to x86 platforms. We build our model in an all-in-one first-order logic and programming system. Our model enables formal verification of CHERI designs and the rapid-prototyping of new CHERI architectural features, and we apply existing industry push-button verification tools and custom heuristics to prove the correctness of CHERI artifacts.

Index Terms—Formal models, operational semantics, theorem proving, ACL2, capabilities, Y86

I. INTRODUCTION

Capability Hardware Enhanced RISC Instructions (CHERI) is a set of architectural features that enables software compartmentalization and more precise and flexible memory protection than traditional memory protection mechanisms [1]. It is designed to mitigate various types of security vulnerabilities, particularly those related to memory safety, via various hardening techniques. Recently, Arm has shipped Morello, a multicore, superscalar, Neoverse N1-based processor, System-on-Chip, and prototype board to academic and industry security experts [2]. These shipped Arm hardware artifacts are fully integrated with CHERI protection features. Microsoft has described CHERI as a “building block for higher-level security abstractions” [3].

At the center of CHERI are capabilities, which are unforgeable tokens in hardware. These tokens describe the range and actions permitted for various kinds of memory accesses. Capabilities provide a way to control and restrict access to memory at a fine-grained level by including information about the base address of the memory range for which access is permitted, the size of the range, and the permissions associated

with that range. One can view capabilities as *fat-pointers*, i.e. pointers containing metadata associated with the memory address to which the pointer value points. In the case of a capability, the metadata are the permissions, memory bounds, and the capability’s object type.

In CHERI systems, *all* memory access is controlled by capabilities. This fundamentally changes how machine code programs are executed, requires additions to instruction set architectures, and modifies architectural state parameters, such as registers, program counters, etc. The CHERI ISA specification technical reports document each of these additional instructions and other architectural contributions and changes. However, the most recent CHERI ISA contains over 500 pages of technical details, motivation, design specifications, algorithms, etc [4].

We present a formal executable model of the Y86 ISA integrated with CHERI, which we call CHERI-Y86. Y86 is an ISA that executes a simplified subset of the x86-64 instruction set [5]. It was originally introduced by Randal Bryant and David O’Hallaron, and is commonly used by hundreds of institutions to teach computer systems.

It is important that CHERI-Y86 faithfully implements the CHERI-ISA specification [4]: every opcode encoding, register-state extension, permission check, and capability-compression routine is lifted directly from the official manual. To help prevent transcription or interpretation errors, we validated our model using an adapted CHERI conformance test suite designed based on the CHERI-x86-64 ISA descriptions, ensuring correct behavior for all non-deprecated capability-aware instructions. We apply our model primarily to the verification of machine code programs that involve capabilities and formally prove the correctness of CHERI architectural design and security features. This includes proofs for all non-deprecated capability-aware instructions, verification of CHERI’s capability-compression scheme, guarded accesses to any state parameter in CHERI-Y86, and explicit formal executable specifications for CHERI methods and algorithms. CHERI-Y86 is publicly available¹.

There are two additional broader motivations behind our development of CHERI-Y86. First, Y86 is similar to the x86 ISA. While CHERI specifications for x86 are not yet fully elaborated, CHERI-Y86 serves as a useful proof-of-concept for building future models and hardware that integrate

¹<https://gitlab.com/cheri-acl2/y86-cheri-capabilities>

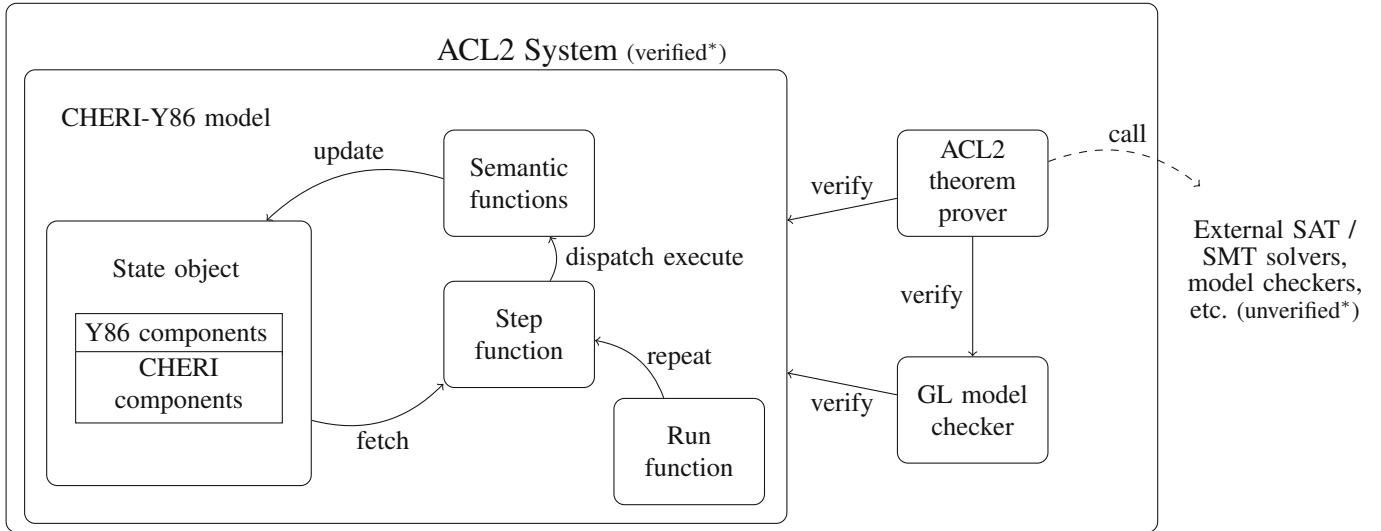


Fig. 1. ACL2 CHERI-Y86 model overview

CHERI with x86. We developed the CHERI-Y86 features in a purposely modular fashion. This is to enable future integration of our model with formal models of the x86 ISA (pending completion of the CHERI x86 specification). This will be invaluable to industry and academic efforts to build highly reliable and secure x86 hardware and low-level software systems. Second, Y86 is much simpler than x86. One of our major goals is to elaborate key CHERI features with our model rather than address a lot of specific architectural details. By using a stripped-down ISA, we emphasize the already numerous architectural features of CHERI without introducing the added burden of an otherwise complicated instruction set. This makes CHERI accessible to students and professionals, and enables the introduction of CHERI features to engineers of other architectures that wish to secure their designs without having to learn the complexities of another platform.

Our model is built in ACL2 [6], a first-order logic and proof system that is fully programmable, extensible, and fast, with support for industry-strength push-button verification tools. Employing ACL2 in building our model of CHERI-Y86 combines the convenience of fully automatic tools and a general purpose theorem prover, enabling users to offload tedious brute-force verification processes and focus more on purpose-driven model and proof design aspects.

II. RELATED WORK

A *capability* is an unforgeable token of authority to access a system object, along with a set of operations that the holder of the capability is permitted to perform on the object. Possession of a capability is sufficient to guarantee that the holder has the indicated access permissions to the object. In theory, the use of capabilities obviates the need for access control lists or similar security structures; entities are provided exactly those capabilities needed to perform their function. This potentially facilitates fine grained access control and enforcement of least privilege, but also necessitates that the system prevent

user programs from forging or modifying capabilities. Such protections may be implemented in hardware or software.

Although several earlier systems included capability-like addressing mechanisms, the term “capability” was first introduced by Dennis and Van Horn [7] in the mid-1960s as part of an hypothetical operating system supervisor for a multiprogramming system. Capabilities addressed a variety of issues in OS security, including sharing and cooperation among processes, protection of processes, and naming of objects [8, p. 41]. They subsequently were adopted by a number of hardware and software systems. Levy’s *Capability-Based Computer Systems* [8] provides a detailed overview of capability-oriented computer system design through the 1980’s. More recently, capabilities have seen adoption in a variety of systems. The CHERI documentation provides an excellent overview [9]. Some notable recent systems implementing capability-based security include: EROS [10] (now CapROS and Coyotos [11]), a capability system that permits creating confined subsystems; and seL4 [12], a formally verified, capability-oriented microkernel.

Formal verification, specifically machine checked proofs of security properties, is especially desirable in the case of capability-based systems in light of claims by Boebert [13] and others that unmodified capability systems cannot enforce fundamental access control properties. Both EROS and seL4 included significant verification efforts. A proof [14] of the EROS confinement properties was carried out by the developers, though it appears that the proof is not machine checked. The seL4 microkernel has been subjected to formal analyses beyond any other comparable system. Extensive proofs [15] of seL4’s access control system and of the functional correctness of the kernel have been carried out using Isabelle/HOL.

Lightweight formal methods have also been deployed to CHERI in particular. CHERI-MIPS and CHERI-RISC-V benefit from a suite of tools that enable their formal specification

in a domain-specific language called Sail [16]. The advantage to this approach is that generating Isabelle specifications from Sail is automatic, and an impressive set of security properties for CHERI-MIPS has been verified in Isabelle [17]. Extensive formal verification efforts behind the Morello architecture also deploy an extensive suite of translation layers, validation suites / test generators, specification languages, and various theorem proving tools (mostly revolving around Isabelle) to ensure vital properties involving capability monotonicity [18]. Similarly, encoding / decoding aspects of CHERI Concentrate have also been verified using Sail’s SMT backend in the CHERIOT work [19]. However, proving properties about these specifications require a translation layer from Sail or other specification languages to the Isabelle theorem prover. Interactions between different systems introduces logical concerns involving soundness. Moreover, while Isabelle can express a wide breadth of mathematics, support for execution and simulation is limited. The work we present in this paper enables the efficient modelling, simulation, and verification of CHERI features for an x86-like ISA all in a single logic.

A subset of the CHERI ISA has also been formalized in the operational semantics approach using Coq [20], [21]. The focus of this work is to verify untrusted machine code programs do not violate certain security properties on a CPU which supports capabilities. We are also interested in verifying machine code programs using our simulator, but our focus with this work is to develop a formal model with formalized CHERI features that gets us closer to existing real-world ISAs.

Formal proofs have become an essential component of industrial hardware development, from the proof of the AMD floating point multiplication unit [22]. All major hardware manufacturers utilize automated reasoning tools to deal with the immense complexity of modern digital circuitry [23].

ACL2 [6], specifically, has been used to prove properties of digital systems, both hardware and software, for decades and is in widespread use in industry [23]. Numerous machine models have been built in ACL2 over the years including the Sun Java Virtual Machine, the Motorola CAP digital processor, and many more. The ACL2 homepage contains an extensive publications list.² Most relevant to the current project are those modelling the x86 ISA [24]–[27]. The ACL2 x86 model is similar to CHERI-Y86 in that they both simulate the behaviour of a machine by way of operational semantic functions which update a state object formalized as an ACL2 single-threaded object. The effect of machine code on both models can be symbolically simulated using Boyer-Moore clock functions and formally verified. An advantage to the ACL2 x86 model is that it is the largest and most comprehensive specification of an x86 machine, to the point where it can be used to boot Alpine Linux. In fact, the Sail specification of the x86 architecture is mechanically translated from the ACL2 specification.

III. FORMALIZED CHERI FEATURES

The underlying CHERI protection feature at the level in which we are interested is the restriction of access to archi-

tectural components (e.g. memory, registers, or code objects) by way of minimizing the permissions available to other components (e.g. executable instructions), thus compartmentalizing and limiting the effects of bugs or other vulnerabilities. Permissions and the scope of permissions are controlled by *capabilities*.

The CHERI ISA specifies two representations of capabilities and we formalize both. *CHERI Concentrate* is the explicit format for the representation of capabilities *in memory*. Capabilities represented abstractly with software-accessible fields (some of which may not be explicitly stored in memory) are called *architectural capabilities*. As part of our formalization, we develop and verify ACL2 functions that allow a user to easily convert between architectural and memory-resident capabilities. The proofs of these conversion functions amount to verifying that converting from architectural capabilities to memory-resident capabilities and back again recovers the information that was originally in the architectural capabilities, and vice-versa. In this section, we describe our formalizations of CHERI architectural capabilities, CHERI Concentrate, and protection features.

A. CHERI Concentrate

System bandwidth, throughput, and memory limitations have led to many iterative modifications to the compressed format of capabilities, requiring the need for encode and decode functions. CHERI uses a compressed format called CHERI Concentrate, which specifies a 64-bit representation for capabilities on 32-bit systems and a 128-bit representation for capabilities on 64-bit systems. We are only interested in 128-bit capabilities, the format is summarized by Figure 2. The format contains:

- a : a 64-bit address;
- p : 16 permission bits including various read, write, and execute permissions;
- o : 18 object type (otype) bits indicating whether and how a capability is sealed;
- I : 1 internal exponent bit determining whether the bounds are in exponent format;
- $t_{[11:3]}$ & t_E : 12 bits for computing the capability’s “top” 14-bit memory address;
- $b_{[13:3]}$ & b_E : 14 bits for computing the capability’s “base” 14-bit memory address.

CHERI Concentrate uses a compressed floating point representation to encode the “top” t and “base” b bounds relative to a capability’s address. It supports two formats: if I is set, then the lowest three bits b_E and t_E of b and t , respectively, are used to represent an exponent at the expense of three bits of precision; otherwise, the exponent is considered to be zero giving back b and t their lower three bits. An interested reader should consult the CHERI ISA [9] and original CHERI Concentrate [28] paper for details.

In addition to formalizing CHERI Concentrate, we also present the formal verification of the encode / decode function properties. The role of the encode / decode functions is to compress / decompress the bounds for the region of memory

²<https://www.cs.utexas.edu/users/moore/publications/acl2-papers.html>.

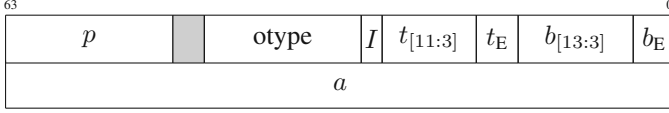


Fig. 2. CHERI Concentrate 128-bit format.

a capability is permitted to access, which are specified by a base address and a length. When added together, the base address and the length form a top address, i.e. the upper bound in memory a capability is permitted to access. Due to compression, some accuracy is lost after encoding the top and base, except in certain circumstances. Let b_0 , ℓ_0 , and t_0 be the base, length, and top before encoding, respectively. Let b_1 , ℓ_1 , and t_1 be the result of encoding and then decoding b_0 , ℓ_0 , and t_0 , respectively. These are the properties we verify:

- 1) $b_0 \geq b_1$ for any b_0 , t_0 , and address;
- 2) $b_0 - b_1 \leq 2^{E+3}$ for any b_0 , t_0 , and address;
- 3) $t_0 \leq t_1$ for any b_0 , t_0 , and address;
- 4) $t_1 - t_0 \leq 2^{E+3}$ for any b_0 , t_0 , and address;
- 5) $b_0 = b_1$ and $t_0 = t_1$ when the lower $E + 3$ bits of b_0 and t_0 are zero;
- 6) $b_0 = b_1$ and $t_0 = t_1$ when $\ell_0 < 2^{12}$.

Here E is a 6-bit value representing an exponent when $\ell_0 \geq 2^{12}$. Properties (1) and (3) ensure that compression does not relax the memory bounds originally intended by the capability, thus ensuring secure memory compartmentalization. Properties (2) and (4) guarantee that any rounding error resulting from compression is not too large. Properties (5) and (6) specify the exact conditions for which the bounds can be recovered exactly. We discuss the verification of these properties and the formalization of CHERI Concentrate in Section VI.

B. Architectural Capabilities

CHERI Concentrate is a format for storing capabilities *in memory*. We make a distinction between *architectural capabilities* and capabilities in memory. Architectural capabilities contain software-accessible fields that are not reflected explicitly by, but can still be inferred from, a given capability's in-memory representation. In our model, architectural capabilities contain the following fields: 1-bit validity tag; 16-bit permissions (perms); 18-bit object type (otype); 64-bit offset; 64-bit base; and 64-bit length. The base is the lower bound of the memory region that can be dereferenced by a capability. Adding length to base gives the upper bound (a.k.a. top) of the memory region that can be dereferenced by a capability. Adding offset to base gives the address when the capability is used as a pointer.

System constraints necessitate compressing the information provided by an architectural capability into the format described by CHERI Concentrate, which is then stored in memory. We formalize a kind of program interface in which numerous functions translate between the human readable architectural capabilities format and the compressed raw bits of CHERI Concentrate. For fields such as perms or otype, simple direct bit-to-integer and integer-to-bit translations suffice. Translating offset, base, and length fields from an architectural

Name	Field	Description
Registers	rgf	16 general-purpose 128-bit registers
RIP	rip	Program counter
ZF	zf	Zero flag
SF	sf	Sign flag
OF	of	Overflow flag
Memory	mem	2^{64} bytes modelled with 2^{24} addresses
MS	ms	Model state, indicates model errors

TABLE I
CHERI-Y86 MACHINE STATE OBJECT

capability to the base, top, and address fields in CHERI Concentrate requires much more sophisticated interface functions and must satisfy the properties discussed in Section III-A

IV. THE ACL2 CHERI-Y86 MODEL

CHERI-Y86 is formalized using an interpreter approach to operational semantics and direct proofs verified using symbolic simulation based on Boyer-Moore clock functions [29]–[31]. A machine state object is defined in the ACL2 logic with field access and update functions. Semantic functions for each instruction in the ISA are implemented using the state object's access and update functions. A step function is defined to handle the fetch-decode-execute cycle, calling the appropriate instruction semantic functions during the execute stage. A run function is defined to take a number n of instructions to be executed and an initial CHERI-Y86 state object, and attempts to execute n calls to the step function.

A. The CHERI-Y86 State Object

We use a *single-threaded object* (stobj) to represent the CHERI-Y86 state in ACL2 [32]. The technical details of stobjs are beyond the scope of this paper, but the upshot is that stobjs enable extremely efficient field updates, and therefore fast execution speeds, by way of destructive memory assignments.

The fields of the CHERI-Y86 state `y86-64` are summarized in Table I. Those familiar with the Y86 architecture will find the CHERI-Y86 state machine similar, but some fields deserve extra comments. The first field is a static array of 16 general purpose registers. These registers are 128-bit in order to accommodate 128-bit capabilities. CHERI systems in general make a distinction between general-purpose registers and *capability registers*. However, the CHERI-x86-64 intends to extend existing general-purpose 64-bit x86 registers to 128-bit in order to store capabilities in the CHERI Concentrate format. We likewise accommodate 128-bit capabilities by extending general-purpose 64-bit Y86 registers to 128-bit. The Y86 model is intended to represent a 2^{64} byte physical memory and we maintain the byte-level semantics specified by the x86 specifications. The model state (MS) is a flag representing the state of the model as a simulator in ACL2 and not part of the original Y86 processor. It is used to signal problems with the model, such as when the CHERI-Y86 processor is halted. While the flag is `nil`, there is no problem with the CHERI-Y86 processor model; otherwise, the MS flag indicates the problem.

Each field of a model can be accessed by passing the CHERI-Y86 object corresponding to the model and invoking

the appropriate field. For example `(rip y86-64)` returns the program counter of `y86-64`. Fields with multiple parameters can be similarly accessed by passing the desired key or address. For example, to access the 0th register `rax`, use `(rgfi 0 y86-64)` or `(rgfi :rax y86-64)`. Updating the fields performed by prefixing `!` to the respective read commands. For example, to reset the program counter of `y86-64` to 0, invoke `(!rip 0 y86-64)`. Similarly, `(!rgfi :rcx #xB0BA7EA y86-64)` updates the 128-bit register `rcx` with 185313258.

B. Symbolic Simulation

Designed for symbolic or functional simulation, our simulator allows users to express machine state and instruction streams in symbolic ACL2 representations, enabling detailed, step-by-step observation of the model’s behavior—including permission checks, capability manipulations, and error signaling. By working with interpretable, symbolic input and output, we lower the barrier for users to explore, reason about, and validate CHERI mechanisms within our x86-like model. While the current implementation does not natively load compiled binaries, extending the simulator to do so would primarily involve building pre- and post-processing layers that translate between binary formats and these symbolic representations. This design choice keeps our core model transparent and amenable to formal verification, while leaving open the possibility of direct binary support in future work.

C. Step & Run Functions

Simulation of machine program execution is performed by a sequence of calls to the CHERI-Y86 step functions. Each step function fetches an instruction from the memory of the CHERI-Y86 state object, decodes it, and then initiates the execute stage. Execution of the instruction is performed by semantic functions, which interpret the intended behaviour of the state under the instruction’s effects and transforms the CHERI-Y86 processor from one state to the next. Modifications to the model state are permitted only if the appropriate capabilities are in place. The step function `y86-step` can be called on a state object `y86-64` with `(y86-step y86-64)`, which returns a new model state object after performing a fetch-decode-execute cycle. Note that this new state object may contain an error flag. Passing the number of steps desired to the `y86` function, e.g. `(y86 y86-64 n)` will attempt to run `y86-step` `n` times. If the run function encounters a state object with the MS flag raised, then `y86` immediately returns the state object. Typically, the MS flag indicates a model error or that a HALT instruction has been encountered. Otherwise, `(y86 y86-64 n)` will return the `y86-64` object in the state after `n` instructions have been executed.

D. CHERI-Y86 Instruction Semantic Functions

CHERI-Y86 instruction semantic functions specify the effect of machine code instructions on the CHERI-Y86 state. Our modelling of the Y86 instructions is comprehensive

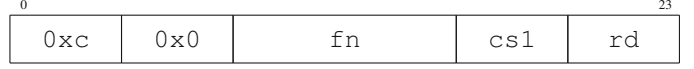


Fig. 3. CHERI-Y86 capability-inspection instruction format.

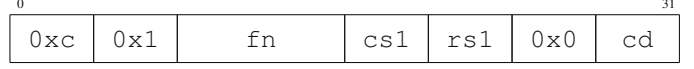


Fig. 4. CHERI-Y86 capability-modification instruction format.

and we implement the CHERI-Y86 equivalent of all non-deprecated instructions that are in the CHERI-x86-64 ISA v8 specifications. Because the latest CHERI-x86-64 specifications are merely a sketch, we take implementation inspiration from the CHERI-RISC-V where necessary.

We develop our own format for CHERI-Y86 capability-aware instructions while adhering to CHERI design goals. Figure 3 contains a visual representation of the CHERI-Y86 *capability-inspection* instruction format. A capability-inspection instruction reads the field of a capability stored in capability register `cs1` and writes the field to register `rd`. All CHERI-Y86 registers have been extended to support storing 128-bit capabilities, so `cs1` simply refers to a general-purpose register as well. We prefix instructions unique to CHERI with `0xc` to indicate they are capability-aware. The `fn` field in Figure 3 contains one of nine possible values since there are nine capability-inspection instructions described in the CHERI-x86-64 specification. Similarly, we’ve implemented all the capability-modification, pointer-arithmetic, and pointer-comparison instructions possible for a Y86-based architecture.

As an illustration, we briefly describe the implementation of GCPERM, a capability-inspection instructions which reads the permissions of a capability and stores them in a register. Algorithm 1 describes the GCPERM instruction with explanations inline. Part of the decode process is handled by the instruction semantic functions, namely the parts that handle reading data contained within the instruction such as source / destination registers, immediate values, memory addresses, etc. The assignments to `pc`, `rArB`, `rB`, and `rA` in Algorithm 1 are a partial consequence of the decode process. Identifying the particular instruction itself is handled by the step function, since it needs to dispatch execute responsibilities to the appropriate semantic function. The capability to be inspected `cs1` is stored in register `rA` in CHERI Concentrate format. The call to `GETCAPABILITY` is our pseudocode way of calling an interface function which decodes `cs1` in CHERI Concentrate format into an architectural capability. Assigning the decoded capability to `cs1` permits us to access the architectural capability fields with functions such as GCPERM, which reads the permissions field. Finally, updates are made to the CHERI-Y86 state object and the state object is returned.

Notice that state objects with the model state flag raised are returned when the instruction attempts to perform an unintended task. In Algorithm 1, this happens when the program counter is at an address in memory that would not

fit the size of the instruction, or when an attempt to read from register 15 is made, which is prohibited in Y86. For more complicated instructions, such as capability-modification instructions, setting the model state flag plays a much larger role. For example, CHERI protection features specify that capabilities should follow a kind of “monotonicity” property, which we will describe in Section V. The upshot is that should an instruction attempt to violate a CHERI architectural security property, then the semantic function will return a state object with a model state flag raised and an error message describing the property violated. Because these security properties are built into the *definition* of the semantic functions, *proving* that our model satisfies the CHERI protection features amounts to unwinding definitions. We also describe our proofs in Section V.

V. VERIFYING BASIC CHERI-Y86 PROPERTIES

In this section, we describe some basic ACL2 features used to ensure the veracity of our CHERI-Y86 model and verify the CHERI protection features within. Rewrite rules are the method by which most ACL2 proofs are discharged. Rewrite rules enable the ACL2 system to replace subterms of a formula with other terms. Usually, rewrite rules are introduced into the ACL2 logic when a theorem is proven. Some theorems are automatically introduced into the logic whenever a new function is defined. For example, the *definition* of the function is a theorem itself. The rewrite rule associated with this theorem enables ACL2 to replace a call to the function with its definition, essentially “expanding” a function. A less trivial example is the theorem that states a newly defined function eventually terminates on all inputs. All functions in ACL2 must be proven to terminate before they are accepted into the logical universe. This is especially important for functions such as `y86` where avoiding the halting problem is desirable. In the case of `y86`, the function is *measured* by the number of steps it is given; that is, either $(\text{y86 } \text{y86-64 } n)$ will terminate after n recursive calls or a model state error will be raised.

Some basic but important theorems of our model are *read-over-write* lemmas. These lemmas state that reading from a written field recovers the value written. For example, Listing 1 contains a theorem which states that writing a value v to register i in the `y86-64` state object, and then reading from register i again results in the same value v . Proving read-over-write lemmas are a good example of using rewrite rules. Expanding the definitions of `rgfi` and `!rgfi` exposes that underneath the `stobj` implementation are array-like data structures with their own read-over-write lemmas. The proof of `rgfi-!rgfi` in Listing 1 can be roughly thought of as the following sequence of rewrites

```

(rgfi i (!rgfi i v y86-64))
→ (nth i (!rgfi i v y86-64))
→ (nth i (update-nth i v <rgfi-array>))
→ v

```

Similarly, there are theorems that describe fields to which no writes are performed. Suppose a write is made to memory

```

(defthm rgfi-!rgfi (equal (rgfi i (!rgfi i v
y86-64)) v))

```

Listing 1. ACL2 read-over-write lemma for registers.

```

(defthm memi-read-through-different-address-!memi
  (implies (and (n64p i) (n64p j) (not (equal i j)))
    (equal (memi i (!memi j v y86-64))
      (memi i y86-64))))

```

Listing 2. ACL2 theorem for memory invariance.

address i . It should follow that the memory at any address other than i is unchanged. Listing 2 contains the ACL2 theorem that states this property, which is discharged via rewriting as well. This property is perhaps even more important as it is a guarantee that contributes significantly to memory safety.

In addition to the read-over-write lemmas just described, there are also three more classes of theorems that together usually form a sufficient theory for proving more desirable theorems about a model:

- 1) write-over-write lemmas: two writes to different memory addresses can be performed in any order while resulting in the same final state object, and if the memory addresses were the same, then reading from the same memory address will result in the value of the most recent write;
- 2) write-over-read lemmas: writing a value that was just read from a memory address back into the same memory address will result in the same initial machine state;
- 3) state well formedness lemmas: writing a valid value to a field in a well formed machine state object results in another well formed state.

Basic protection features of CHERI capabilities are also proven using core ACL2 theorem proving features. For example, CHERI restricts capabilities that arise from other capabilities to an monotonicity property; that is, there is a partial order $\leq$ on capabilities, and if a capability-modification instruction is given a capability c_1 that results in a new capability c_2 , then we must have $c_2 \leq c_1$. For example, consider `CSETBOUNDS`, a CHERI-Y86 instruction that takes a capability c_1 and register r_2 , and creates a new capability c_2 with bounds based on the address of c_1 and the value in r_2 (see Figure 4 for context). All other fields of c_2 are the same as c_1 . In this case, we have $c_2 \leq c_1$ iff the bounds of c_2 are contained within the bounds of c_1 , i.e. $\text{GETBOUNDS}(c_2) \subseteq \text{GETBOUNDS}(c_1)$. This is a reasonable CHERI protection feature; a capability should not be able to create a capability with greater memory access than its own. The way we prove this is simple:

- 1) initialize a well formed valid (no MS flag set) CHERI-Y86 state;
- 2) *symbolically* simulate one fetch-decode-execute cycle using `CSETBOUNDS`;
- 3) prove $\text{GETBOUNDS}(c_2) \subseteq \text{GETBOUNDS}(c_1)$.

Performing step (3) amounts to applying rewrite rules the definition of `CSETBOUNDS`. Recall the definition of `GCPERM` in Algorithm 1. When the instruction semantic

Algorithm 1 High-level description of CHERI-Y86 GCPERM instruction implementation.

procedure GCPERM(y86-64)

pc ← RIP(y86-64)

if pc < 2⁶⁴ - 4 **then** **return** !MS(y86-64, “PC too large”)

rArB ← READMEM(pc + 2, y86-64)

rB ← rArB & 2⁴rA ← SHIFTRIGHT(rArB, 4) & 2⁴ - 1**if** rA = 15 ∨ rB = 15 **then** **return** !MS(y86-64, “Prohibited register”)

cs1 ← GETCAPABILITY(y86-64, rA)

cPerms ← GETPERM(cs1)

y86-64 ← !RGFI(rB, cPerms, y86-64)

y86-64 ← !RIP(pc + 3, y86-64)

return y86-64

▷ Get PC from state

▷ Check if PC is too large

▷ Return state w/ MS flag set & error description

▷ Get the byte specifying src & dest registers

▷ Get the dest register by masking upper 4 bits

▷ Get the src register by shifting & masking

▷ Check if prohibited register specified

▷ Return state with MS flag set

▷ Get & decode capability from src register

▷ Get permissions from capability

▷ Store permissions in dest register & update state

▷ Update PC in state

▷ Return updated state

function attempts to perform a prohibited action, the function returns a state object with the MS flag raised. Similarly, the semantic function for CSETBOUNDS returns a state object with the MS flag raised should c_2 be assigned bounds outside of c_1 's bounds. Therefore, if a valid CHERI-Y86 state object is returned after the fetch-decode-execute cycle is performed in step (2), then we must have $\text{GETBOUNDS}(c_2) \subseteq \text{GETBOUNDS}(c_1)$ and thus $c_2 \leq c_1$.

VI. VERIFYING CHERI CONCENTRATE

Our approach to proving the properties in Section III-A makes heavy use of the symbolic simulation framework GL with case-splitting [33], [34]. GL supports model checking with both binary decision diagrams (BDDs) as ACL2 symbolic objects and external SAT solvers. We use BDDs as the back-end engine for GL. While case splits were necessary for some of the larger proofs, we found that GL BDDs were sufficient for our verification needs. Moreover, GL's BDD proof procedure is verified in ACL2, alleviating any extra soundness concerns that may arise from calling external tools.

The ACL2 proofs of the encode / decode properties 5-6 are straightforward applications of GL's default symbolic simulation event `def-gl-thm`. To illustrate the use of GL, recall property 6 states

6) $b_0 = b_1$ and $t_0 = t_1$ when $\ell_0 < 2^{12}$.

Listing 3 contains the ACL2 theorem for property 6, i.e. the top and base bounds are exactly recoverable when $\ell_0 < 2^{12}$. The expression associated with the `:hyp` key

```
(def-gl-thm decode-encode-equal-small-seg
:hyp (and (valid-addr-p addr base len)
          (valid-b-l-p base len)
          (< len (expt 2 *TW*)))
:concl (equal (decode-compression
               (encode-compression len base)
               addr)
              (bounds (+ len base) base))
:g-bindings '((base , (gl::g-int 0 3 65))
              (len , (gl::g-int 1 3 66))
              (addr , (gl::g-int 2 3 65))))
```

Listing 3. ACL2 theorem for property 6 using GL.

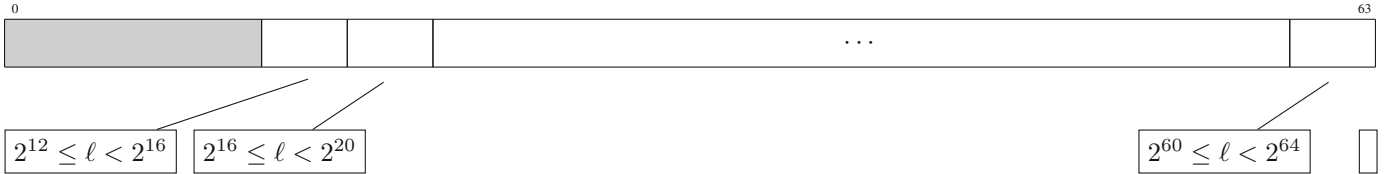
is the GL method of indicating the hypotheses of the theorem. In this case, `(valid-addr-p addr base len)` and `(valid-b-l-p base len)` simply check that `addr`, `base`, and `len` are appropriately “typed.” The hypothesis `(< len (expt 2 *TW*))` just checks that $\ell_0 < 2^{12}$. The key `:concl` denotes the conclusion of the theorem. For property 6, this simply states that decoding the encoded `base` and `len` with respect to an `addr` returns the well-formed bounds given by exactly the original `base` and `len`. The `:g-bindings` are shape specifications which assign the individual bits of `base`, `len`, and `addr` to BDD variables. The assignment of `base` to `gl::g-int 0 3 65` indicates the bits of `base` are assigned to BDD variables indexed 0, 3, 6, ..., 192 = $3 \times (65 - 1)$. Similarly, the bits of `len` are assigned to BDD variables indexed 1, 4, 7, ..., 193 and the bits of `addr` assigned to BDD variables indexed 2, 5, 8, ..., 194. One sign bit is required to satisfy the integer type but is redundant. The “mixed” order of the variables are for performance purposes. Under the hypotheses and shape specifications, the “domain” of the theorem is finite (i.e. `base`, `addr`, and `len` are 64-bit) and thus the theorem is amenable to verification via exhaustive symbolic execution of the functions (i.e. the `encode-compression`, `decode-compression`, `+`, and `bounds` functions) in the theorem under the BDD expressions. Moreover, the hypothesis restricts ℓ to $\ell < 2^{12}$, giving it relatively few possible values relative to its full 64-bit width (see Figure 6 for a visual representation), making the problem tractable for automated procedures. GL performs this symbolic simulation and readily proves the desired theorem.

In contrast to properties 5-6, properties 1-4 all required parameterized case splitting. Listing 4 is the ACL2 theorem of the encode / decode property 1, which states

1) $b_0 \geq b_1$ for any b_0 , t_0 , and address.

Note that the case where $\ell < 2^{12}$ is already handled by property 6, so we may restrict ourselves to when $\ell \geq 2^{12}$. Instead of attempting the proof while assuming the full 64-bit width of the three integers of interest, we split the domains of `base`,

```
(def-gl-param-thm decode-encode-b-bound-len>2^12
  :hyp (and (valid-addr-p addr base len)
            (valid-b-l-p base len)
            (<= (expt 2 *tw*) len))
  :concl (<= (bounds->base (decode-compression (encode-compression len base) addr))
            base)
  :param-bindings
  `(((low 12) (high 16)) , (gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
    ((low 16) (high 20)) , (gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
    ...
    ((low 64) (high 65)) , (gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65)))))
  :param-hyp (and (<= (expt 2 low) len) (< len (expt 2 high)))
  :cov-bindings (gl::auto-bindings (:mix (:nat base 65) (:nat len 65) (:nat addr 65))))
```



len, and addr into 4-bit “intervals,” effectively performing individual symbolic simulations for when the length ℓ satisfies

$$2^L \leq \ell < 2^H$$

for $L \in \text{LOW} := \{4n : n \in \mathbb{N} \wedge 3 \leq n \leq 16\}$ and $H \in \text{HIGH} := \{4n : n \in \mathbb{N} \wedge 4 \leq n \leq 16\} \cup \{65\}$, i.e. $2^{12} \leq \ell < 2^{16}$ and $2^{16} \leq \ell < 2^{20}$ and $2^{20} \leq \ell < 2^{24}$ and so on. In order to perform case splitting soundly, we must also show that our choice of splits for ℓ “covers” the intended domain of ℓ , i.e.

$$\begin{aligned} & \{\ell \in \mathbb{N} : 2^{12} \leq \ell < 2^{65}\} \\ & \subseteq \{\ell \in \mathbb{N} : L \in \text{LOW} \wedge H \in \text{HIGH} \wedge 2^H \leq \ell < 2^H\}. \end{aligned}$$

Visually, this is represented by Figure 5. Note that the hypothesis stipulates that `len` is 64-bit, but we use an extra bit in the GL proof to make the bounds and coverage proof easier to state. The `:hyp` and `:concl` keys in Listing 4 indicate the hypothesis and the conclusion, respectively, similar to Listing 3. The key `:param-hyp` indicates *how* the case splitting should be performed; here, `len` is given various upper and lower bounds determined by different values of `low` and `high`. The values of `low` and `high` are given by the `:param-bindings`. GL “auto-bindings” are a convenient macro for defining the same shape specifications as in Listing 3 with the same “mixed” BDD variable ordering. In order for a proof by cases to be sound, we must also prove that the cases cover the domain of the theorem, which is why shape specifications are also provided to the `:cov-bindings` key.

Case splits for the CHERI Concentrate properties 2-4 are the same. Case splitting into smaller bit “intervals” would discharge the proofs faster at the expense of increasingly verbose user-provided hints. Case splitting being an effective approach suggests that the complexity of encoding and decoding does not scale with the length of the memory regions. Indeed, this accords with the design principles of CHERI Concentrate: “encoding efficiency, minimize delay of pointer arithmetic, and eliminate additional load-to-use delay” [28]. On the other hand, properties 5-6 required no case splitting largely because the encode / decode logic is greatly simplified when $\ell_0 < 2^{12}$, and alignment requirements reduce the number of variable bits.

VII. CONCLUSIONS AND FUTURE WORK

We presented a formal model for the concurrent simulation and verification of CHERI architectural features. We extend Y86-64 with CHERI features because we intend to perform the same analysis on the x86-64 ISA augmented with CHERI features; however, the full specifications for CHERI-x86-64 have not yet been developed. Our model is built using the ACL2 programming language and verified with the ACL2 theorem prover. Function definitions double as functional specifications; many fundamental properties are verified via function expansion. For bit-twiddling proofs where hand-guiding the theorem prover was too cumbersome, we offload the work to GL, a fully ACL2 verified model checker with support for ACL2 BDDs. Our model comprises 9,856 lines of ACL2, including 2,446 lines of proof scripts. Key-property proofs typically completed in about two minutes each.

There are two major directions for future work. First, we intend to model CHERI-x86-64. CHERI-x86-64 is still in development; there is an opportunity to leverage existing ACL2 frameworks toward the new frontier of CHERI systems. Second, our model can be used to verify machine code programs involving CHERI instructions. In this paper, we verified straightline programs involving capability-aware instructions to analyse the protection features of CHERI capabilities. We leave the verification of longer capability-aware machine code programs for the future.

REFERENCES

- [1] R. N. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, “An introduction to CHERI,” University of Cambridge Computer Laboratory, Technical Report 941, September 2019. [Online]. Available: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf>
- [2] R. Grisenthwaite. (2022) Morello research program hits major milestone with hardware now available for testing. [Online]. Available: <https://www.arm.com/company/news/2022/01/morello-research-program-hits-major-milestone-with-hardware-now-available-for-testing>
- [3] S. Amar. (2022) An armful of CHERIs. [Online]. Available: https://msrc.microsoft.com/blog/2022/01/an_armful_of_cheris/
- [4] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, F. A. Fuchs, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, “Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 9),” University of Cambridge Computer Laboratory, Technical Report 987, September 2023. [Online]. Available: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf>
- [5] R. E. Bryant and D. R. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, 3rd ed. USA: Pearson, 2015.
- [6] M. Kaufmann and J. Moore, “An industrial strength theorem prover for a logic based on Common Lisp,” *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, April 1997.
- [7] J. B. Dennis and E. C. V. Horn, “Programming semantics for multiprogrammed computations,” *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, March 1966.
- [8] H. M. Levy, *Capability-Based Computer Systems*. Bedford, MA: Digital Press, 1984.
- [9] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, “Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 8),” University of Cambridge Computer Laboratory, Technical Report 951, October 2020. [Online]. Available: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf>
- [10] J. Shapiro, J. Smith, and D. Farber, “EROS: A fast capability system,” in *Proceedings of the Seventeenth Symposium on Operating System Principles*. ACM, 1999, pp. 170–185.
- [11] J. Shapiro and J. Adams, “Coyotos microkernel specification, Version 0.6+,” <https://hydra-www.ietfng.org/capbib/cache/shapiro:coyotospec.html>, 2007.
- [12] T. Murray, D. Matchuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, “seL4: From general purpose to a proof of information flow enforcement,” in *Symposium on Security and Privacy*. IEEE, 2013, pp. 415–429.
- [13] W. Boebert, “On the inability of an unmodified capability machine to enforce the *-property,” in *Proceedings of the 7th DoD/NBS Computer Security Conference*, September 1984, pp. 291–293.
- [14] J. Shapiro and S. Weber, “Verifying the EROS confinement mechanism,” in *Proceedings of the 2000 Symposium on Security and Privacy*. IEEE, 2000, pp. 166–176.
- [15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal verification of an OS kernel,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. New York: Association for Computing Machinery, 2009, p. 207–220.
- [16] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, “ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: <https://doi.org/10.1145/3290384>
- [17] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell, “Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1003–1020.
- [18] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell, “Verified security for the morello capability-enhanced prototype arm architecture,” in *Programming Languages and Systems*, I. Sergey, Ed. Cham: Springer International Publishing, 2022, pp. 174–203.
- [19] S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson, and H. Xia, “Cheriot: Complete memory safety for embedded devices,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 641–653. [Online]. Available: <https://doi.org/10.1145/3613424.3614266>
- [20] A. L. Georges*, A. Guéneau*, T. Van Strydonck, A. Timany, A. Trieu*, D. Devriese, and L. Birkedal, “Cerise: Program verification on a capability machine in the presence of untrusted code,” *J. ACM*, vol. 71, no. 1, Feb. 2024. [Online]. Available: <https://doi.org/10.1145/3623510>
- [21] T. Van Strydonck, A. L. Georges, A. Guéneau, A. Trieu, A. Timany, F. Piessens, L. Birkedal, and D. Devriese, “Proving full-system security properties under multiple attacker models on capability machines,” in *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, ser. 2022 IEEE 35th Computer Security Foundations Symposium (CSF), Haifa, Israel, Aug. 2022. [Online]. Available: <https://inria.hal.science/hal-03826851>
- [22] J. Moore, T. Lynch, and M. Kaufmann, “A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm,” *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 913–926, September 1998.
- [23] W. A. Hunt, Jr., M. Kaufmann, J. S. Moore, and A. Slobadova, “Industrial hardware and software verification with ACL2,” in *Verified Trustworthy Software Systems: Philosophical Transactions A*, vol. 374, P. Gardner, P. O’Hearn, M. Gordon, G. Morrisett, and F. Schneider, Eds. Royal Society Publishing, 2017.
- [24] S. Goel, “Formal verification of application and system programs based on a validated x86 ISA model,” Ph.D. dissertation, University of Texas at Austin, 2016.
- [25] A. Coglio and S. Goel, “Adding 32-bit mode to the ACL2 model of the x86 ISA,” in *15th International Workshop on the ACL2 Theorem Prover and its Applications*, 2018.
- [26] S. Goel, W. A. Hunt, Jr., and M. Kaufmann, “Engineering a formal, executable x86 ISA simulator for software verification,” in *Provably Correct Systems (ProCos)*, 2017.
- [27] S. Goel and W. A. Hunt, Jr., “Automated code proofs on a formal model of the x86,” in *Automated Code Proofs on a Formal Model of the x86 (VSTTE)*, 2013.
- [28] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, D. Chisnall, B. Davis, K. Gudka, N. Filard, A. Markettos, M. Roe, P. Neumann, R. Watson, and S. W. Moore, “CHERI concentrate: Practical compressed capabilities,” *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, October 2019.
- [29] S. Ray and J. S. Moore, “Proof styles in operational semantics,” in *Formal Methods in Computer-Aided Design*, A. J. Hu and A. K. Martin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 67–81.
- [30] S. Ray, W. A. Hunt, J. Matthews, and J. S. Moore, “A mechanical analysis of program verification strategies,” *Journal of Automated Reasoning*, vol. 40, pp. 245–269, 2008. [Online]. Available: <https://api.semanticscholar.org/CorpusID:532016>
- [31] “Mechanized operational semantics,” <https://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html>, accessed 2024-04-15.
- [32] “Stobj,” https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2___STOBJ, accessed 2024-04-15.

- [33] S. Swords and J. Davis, “Bit-blasting ACL2 theorems,” in *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, November 3-4, 2011, ser. Electronic Proceedings in Theoretical Computer Science, D. Hardin and J. Schmaltz, Eds., vol. 70. Open Publishing Association, 2011, pp. 84–102.
- [34] S. Swords, “Term-level reasoning in support of bit-blasting,” in *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, May 22-23, 2017, ser. Electronic Proceedings in Theoretical Computer Science, A. Slobodova and W. Hunt, Jr., Eds., vol. 249. Open Publishing Association, 2017, pp. 95–111.

A Method for the Verification of Memory Management Software in the Presence of TLBs

Yahya Sohail 

The University of Texas at Austin
Austin, TX, United States of America
yahya@yahyasohail.com

Warren A. Hunt, Jr. 

The University of Texas at Austin
Austin, TX, United States of America
hunt@cs.utexas.edu

Abstract—The correct management of translation lookaside buffers (TLBs) by memory-management software is critical to the security of modern computer systems. We develop a methodology for reasoning about software that modifies address translations executing on machines with a TLB, enabling the verification of the security-critical, memory-management code in operating system kernels at the binary level. We automate proving that the TLB does not contain entries that are inconsistent with the page tables which allows us to reduce the problem of reasoning about address translation in the presence of a TLB to the problem of reasoning about address translation by walking page tables. Our technique is independent of any particular ISA or TLB microarchitecture. To demonstrate the effectiveness of our approach, we add a TLB to the ACL2 model of the x86 ISA, implement this reasoning technique in the ACL2 theorem prover, and verify a page-table-altering program called Zero-Copy that copies a page of data in the virtual-address space by modifying the page tables. We were able to reuse and update substantial portions of a previous version of the proof of correctness that was performed with a version of the x86 ISA model lacking a TLB, confirming the effectiveness of our reduction-based method.

Index Terms—Software Verification, Caching, Address Translation, x86, ACL2.

I. INTRODUCTION

Contemporary microprocessors provide address-translation mechanisms to enable operating system kernels to implement process isolation and other desirable security properties. Without address translation, any running process could modify the memory of any other process or even the kernel. This would make it impossible for operating systems to implement process isolation or differentiate in any way between the permissions of different processes running on the system. The most commonly employed address-translation mechanism is paging, and paging-enabled microprocessors generally implement one or more address-translation caching mechanisms since paging would be prohibitively slow without them.

Since memory-management software is crucial to computer security, its verification has drawn the interest of the formal-methods community, and paging semantics have been specified formally for several ISAs. However, the behavior of memory-management software running on machines with address translation in the presence of address-translation caches is less well studied using formal methods. These caches have architecturally visible behavior, and memory-management software must correctly manage the contents of

such caches to implement the security properties expected of modern operating systems. Thus, it is not sufficient to verify memory-management software using an ISA model lacking models of address-translation caches to ensure that the software will execute correctly on a machine with address-translation caches.

It is challenging to verify memory management software in the presence of a translation lookaside buffer (TLB), a common type of address-translation cache, because ISAs allow much flexibility in their implementation. On most ISAs, a TLB may cache any address translation that has ever been valid unless its corresponding TLB entry has been explicitly invalidated by software since it was last valid in the page tables. This is difficult to reason about because determining whether there may exist a TLB entry for a given translation may require considering the state of the page tables far before the current point in program execution. Furthermore, the page tables themselves form a tree structure in memory which can be very large, making the consideration of many previous states of the page tables even more difficult.

Our primary contribution is an ISA and microarchitecture-independent methodology for reducing the problem of reasoning about software in the presence of a TLB to the better studied problem of reasoning about software on a machine with paging but no TLB. This is not possible in general, due to TLBs having architecturally-visible behavior, but it is possible when the software correctly invalidates stale entries that may be present in the TLB after updating page-table entries. Fortunately, considering only this case is sufficient for verifying memory-management software because invalidating potentially stale TLB entries before they can be used to perform a translation is a property expected to be maintained by correct memory-management software. Our technique is largely automated and allows the user to easily reuse large parts of proofs about memory-management software that were performed assuming there is no TLB.

We implement our technique using the ACL2 theorem prover [1] on the ACL2, x86-ISA model [2] that we modified by adding a TLB. To our knowledge, this model is the most complete formal model of the x86 ISA; it is sufficiently complete to boot Linux and to run multi-programmed Linux programs. To demonstrate the efficacy of our reasoning strategy, we have used our implementation of the technique on

the x86-ISA model to verify a memory-management program, called Zero-Copy. The Zero-Copy program copies a 1 GB page in the virtual-address space by modifying the page tables to have both the source and destination memory regions translate to the same physical address; this avoids the need to actually copy the data in memory. We were able to reuse much of a correctness proof [2] written for a previous version of the ACL2, x86-ISA model that lacked a TLB.

Our implementation of the technique—including the ISA model, the proofs described here, and documentation—are publicly available as part of the ACL2 Community Books [3] under the 3-clause BSD license. The documentation [4] is available online in XDOC [5], ACL2’s documentation system, and the source, including theorems and proofs, is distributed with ACL2 [1], in the `books/projects/x86isa` sub-directory of the ACL2 source tree available on Github at <https://github.com/acl2/acl2>.

We begin in Section II by reviewing related work. In Section III, we summarize relevant background information about address translation, address-translation caches, and their architecturally visible behavior. In Section IV, we describe our technique to reduce reasoning about software on a machine with a TLB to address translation on a machine without a TLB. In Section V, we describe how we added a TLB to our ACL2 x86-ISA model, and we implemented our reasoning technique in the ACL2 theorem prover. Then, in Section VI, we apply this work to verify the Zero-Copy binary program. Finally, in Section VII, we conclude and describe how this work may be employed in the future to verify operating-system, memory-management code.

II. RELATED WORK

Modern formal ISA models include many details and subtle features, like segmentation, paging, TLB definitions, privilege levels, device configuration, and I/O. With such features, the formal-methods community is able to analyze supervisor-level, binary code that performs management of machine resources. This has resulted in ambitious projects to build formally verified operating systems, like seL4 [6], [7]. However, reasoning about the effects of caching on address translation has continued to prove difficult, so the correct management of the TLB is generally assumed when verifying software. For example, this is the approach employed by the seL4 verification effort.

In 2015, Goel et al. developed the ACL2-based, x86-ISA model [2] which we extend and employ to demonstrate our reasoning technique. This ACL2-based, x86-ISA model included a full specification of segmentation and paging in 64-bit mode, but it lacked interrupts and I/O peripherals. Goel used this model to verify the Zero-Copy program that copies data in the virtual address space by modifying address translations in the page tables, demonstrating that it could be used to reason about memory management software. However, this model did not capture the semantics of TLBs or any other address translation caches.

Syeda and Klein describe [8], [9] reasoning about a TLB model that they added to the Fox and Myreen Isabelle/HOL model [10] of the user-level, 32-bit Arm v7 ISA [11]. This extension concerned only the memory system, and their model did not sufficiently specify all of the ISA and memory semantics necessary to boot an operating system. They describe how they translate their model into more abstract memory models, such as a model where the TLB contains all of the address translation information for a particular address-space identifier (ASID). This approach was shown to be logically consistent with their lowest-level memory model.

Syeda and Klein proposed various abstractions of the TLB, and their work argues formally about cache models (and specifically, about the TLB). Their memory-management model captures the essential concepts required to support the ARM v7 [11] virtual-memory system. Their suite of memory models are elegant, and those models expose the essential issue of reasoning about programs that interact with the memory-management system. However, the ISA model lacks the detail required to support the booting of Linux or other operating systems on their ARM v7 model, making it difficult to gain confidence in the model’s accuracy. Furthermore, their work was concerned with only the ARM v7 ISA, and they did not generalize their approach to reasoning about address translation to other ISAs.

In contrast, the work presented in this paper is an ISA and TLB microarchitecture independent method for reasoning about address translation with TLB caches and demonstrated on an x86-ISA model complete enough to boot Linux and run Linux programs. We have validated this model by booting Linux, running user programs under Linux, running program-matically generated tests, and cosimulation. By demonstrating our reasoning strategy on this model, we show that our technique can scale to the challenges of reasoning with large ISA models that capture the semantics and complexities of modern ISAs. Our reduction-based approach also simplifies updating proofs that were done with ISA models that lacked a TLB to accommodate TLB behavior, which we demonstrate by reusing substantial parts of Goel’s Zero-Copy correctness proof, providing a path for updating existing software correctness proofs to account for TLB behavior.

III. BACKGROUND

In a microprocessor that performs address translation, programs execute load and store instructions on memory locations specified by virtual addresses that are mapped, using a function specified by supervisor-level software, into physical addresses by a memory-management system. When address translation is successful, virtual addresses are translated into physical addresses which are used to access memory; otherwise, memory access is denied, and an exception is signaled. Most modern processors that support address translation use *paging* as the mechanism to specify the translation function. Paging allows supervisor software to specify the translation function for each *page*, a naturally aligned block of memory with a size that is

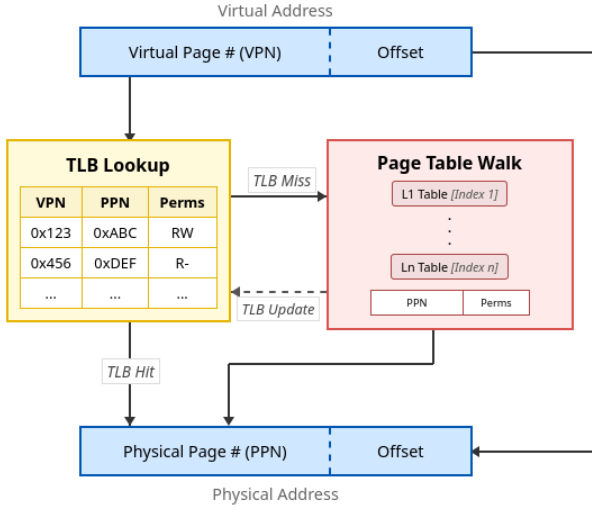


Fig. 1. The virtual address (top) is broken into two parts, the virtual-page number and the offset. The virtual-page number is used to index into the TLB (left). If there is a TLB hit, the VPN is found in the TLB and the corresponding physical page number is returned. Otherwise, a TLB miss occurs and the processor performs a page-table walk (right). The returned PPN and permissions from the page-table walk are used to update the TLB. The offset is concatenated onto the PPN to construct the physical address (bottom).

a power of 2, using memory-resident tables, known as *page tables*, that form a tree structure.

When a microprocessor needs to refer to the page tables to translate an address, it traverses the tree in a process known as a *page-table walk*. This is done by using the high-order bits of the virtual address, called the virtual-page number, to pick a subtree at each level of the page tables until the processor arrives at a leaf-page table entry. As it traverses the page tables, the processor collects information about the permissions of the page that are used to determine whether the processor should raise an exception, and the leaf-page table entry contains the physical page number corresponding to the virtual page number. If the access is allowed and the processor does not raise an exception, the processor translates the virtual address by concatenating the physical-page number to the page offset, the low-order bits of the virtual address that are not part of the virtual page number.

Without caching, paging is prohibitively slow because each translation requires multiple memory accesses. Consequently, microprocessors employ a number of caches to minimize the number of memory accesses necessary for translating addresses. Of these caches, the most common is the translation-lookaside buffer (TLB) that maintains an associative mapping from virtual-page numbers to their corresponding permissions information and physical page number as shown in Figure 1. Processors do not automatically invalidate TLB entries when corresponding page-table entries are updated since doing so would be prohibitively expensive. Thus, it is possible that a TLB contains *stale* entries, entries that do not agree with the page tables. To address this issue, ISAs specify how

```
CLC                                (!rip (+ 2 (rip x86)))
STC                                (!f1gi :cf 1 x86))
```

Fig. 2. On the left is an x86 assembly program that first clears and then sets the carry flag. On the right is the result of symbolically executing the program on the ACL2 x86 model using the ACL2 rewriter under certain preconditions. It has been lifted into an expression in the ACL2 logic that is a composition of two operations: the first, `f1gi`, that updates the flag register—in this case, setting the carry flag to 1, and the second, `rip`, that updates the instruction pointer—in this case by adding two to the current instruction pointer.

software may invalidate TLB entries; for example, in the x86 architecture, executing the instruction `INVLPG` invalidates any TLB entries associated with a specified virtual address and changing the value of the PG bit in the `cr0` register from 1 to 0 invalidates all TLB entries [12]. It is the responsibility of supervisor-system software to employ invalidation operations to invalidate stale TLB entries after updating the in-memory page tables.

While ISAs require that certain entries are invalidated by certain operations, they generally allow implementations to invalidate additional entries whenever they choose. They also allow implementations to perform speculative page table walks and choose whether or not to update address-translation caches after a page-table walk, speculative or otherwise, as they see fit. Thus, any translation which has been valid in the page tables and has not had a corresponding TLB entry explicitly invalidated since it was last valid in the page tables may or may not have a corresponding TLB entry. This makes it difficult to reason about what entries may be present in the TLB at a given point in time.

IV. OUR TECHNIQUE

The primary difficulty when reasoning about address translation with caching concerns the incomplete nature of ISA-level specifications. ISAs often describe weak guarantees about TLB contents, so the behavior of address translation in regions with stale TLB entries is not specified fully. Therefore, software should not rely on the results of address translation in such regions. Consequently, we reason about an address translation first by proving that there are no stale TLB entries corresponding to a desired translation or equivalently that the address translation is *TLB consistent* with respect to the machine state. In this case, the result of the translation is the same regardless of whether a TLB entry or the page tables are used to perform the translation. Thereafter, we can reason about the result of the translation by consulting the page tables.

To help prove that a *translation*—that is a tuple of the virtual address, access type (i.e., read, write, or execute), and the values of various processor control bits that affect address translation—is TLB consistent, we have developed a theory of how address translation is affected by different operations on the machine state. Our model of a machine is a function that takes an initial machine state and a number of instructions to execute as input and returns the machine state after executing the given number of instructions. We can execute this function

symbolically via rewriting under the assumption that the initial machine state contains a given program’s machine code in the machine’s memory at the address given by the instruction pointer. The result of symbolic execution is an expression in the logic that captures the effect of the program on the machine state as shown in Figure 2.

For the following discussion, we define an *operation* to be a function that may appear in the expression that results from symbolically executing a program using a machine model. In Figure 2, the functions `!rip` and `!flgi` are operations. While `!rip` and `!flgi` each only update a single field of the machine state, an operation can perform an arbitrarily complex update to the state of the machine. Any function that returns a new machine state given some parameters and the previous machine state may be an operation. There are two sets of translations associated with every operation that are relevant to address-translation caching behavior:

- *Maintained Translations* — These are translations that are TLB consistent after the operation if they were consistent before the operation. These correspond to translations whose corresponding page-table entries and TLB entries were not updated by the operation.
- *Consistent Translations* — These are translations that are guaranteed to be TLB consistent after the operation is performed, regardless of whether they were TLB consistent before the operation. These translations correspond to translations that have had any corresponding TLB entries invalidated by the operation.

For example, a “load 0 into a general-purpose register” operation’s set of maintained translations is all translations and its set of consistent translations is the null set because loading a general-purpose register does not affect the TLB or page tables. On the other hand, an “invalidate all TLB entries” operation has all translations as its set of consistent translations since invalidating all TLB entries will necessarily invalidate all stale TLB entries.

For each operation, we prove theorems that show which translations, possibly in terms of the parameters to the operation and the previous state, are maintained and consistent. We can then use these theorems to show that a given translation is consistent after a program is executed. Showing that a given translation is in the set of consistent translations of the outer-most operation in an expression for the state of the machine proves the translation is TLB consistent in the state given by that expression. Showing that a translation is in the set of maintained translations of the outer-most operation in an expression allows us to “peel away” the outer-most operation and reduce the proof to showing that the translation is consistent in the state prior to the application of the outer-most operation. We can then recursively attempt to prove the translation is consistent in the prior state.

Without loss of generality, in the following discussion, we assume operations have a single parameter because if an operation has multiple parameters, one can define an equivalent operation with a single parameter that is a tuple of the original

Algorithm 1 PROVE-CONSISTENT(t, k, h)

```

if  $k = 0$  then
  if PROVE( $t$  is consistent in  $q_0$  under hypotheses  $h$ ) then
    return true
  else
    return fail
  end if
end if
if PROVE( $t \in C_{i_k}(p_k, q_{k-1})$  under hypotheses  $h$ ) then
  return true
else if PROVE( $t \in M_{i_k}(p_k, q_{k-1})$  under hypotheses  $h$ ) then
  return PROVE-CONSISTENT( $t, k - 1, h$ )
else
  return fail
end if

```

operation’s parameters. Then, all programs on all machines lift into a sequence of operations that looks like

$$\begin{aligned}
 q_1 &\leftarrow \mathcal{O}_{i_1}(p_1, q_0) \\
 q_2 &\leftarrow \mathcal{O}_{i_2}(p_2, q_1) \\
 &\vdots \\
 q_n &\leftarrow \mathcal{O}_{i_n}(p_n, q_{n-1})
 \end{aligned}$$

where $\mathcal{O}_i$ are the operations of the machine, p_i are parameters to the operations (which may be dependent on the initial machine state), q_0 is the initial state, and q_n is the final state. Let $M_i(p, q)$ and $C_i(p, q)$ be the sets of translations that are maintained and consistent respectively by operation $\mathcal{O}_i(p, q)$. Then we can show translation t is TLB consistent with respect to q_k under hypotheses h using Algorithm 1, PROVE-CONSISTENT(t, k, h), which returns either true indicating that t is consistent with respect to q_k , or fails (in which case it is inconclusive). The PROVE-CONSISTENT algorithm assumes we have some procedure PROVE, for example the ACL2 prover, that can be used to attempt to prove a conjecture under given hypotheses. This procedure either succeeds in proving the conjecture and returns true, indicating the conjecture is a theorem, or it fails, returning false, in which case the conjecture may or may not hold true. In practice, we expect the base case that attempts to prove t is consistent in q_0 succeeds due to a hypothesis in h that states t is TLB consistent in q_0 .

We demonstrate the correctness of PROVE-CONSISTENT(t, k, h), that is PROVE-CONSISTENT(t, k, h) returning true implies t is consistent in q_k , by a simple inductive argument that inducts over k . The base case is when $k = 0$. If we can prove t is consistent in q_0 , it must be consistent in $q_k = q_0$, and we return true; otherwise, we fail, which is inconclusive. For $k > 0$, if we can prove $t \in C_{i_k}(p_k, q_{k-1})$, we return true, and t must be consistent in q_k because it is in the set of consistent translations of $\mathcal{O}_{i_k}(p_k, q_{k-1})$. If we can prove $t \in M_{i_k}(p_k, q_{k-1})$, we return PROVE-CONSISTENT(p_k, q_{k-1}, h), and this is correct because if the recursive call returns true, t must

be consistent since it is in the set of maintained translations of $\mathcal{O}_{i_k}(p_k, q_k)$ and, applying the inductive hypothesis, $\text{PROVE-CONSISTENT}(p_k, q_{k-1}, h)$ returning true implies t is consistent in q_{k-1} . Finally, if we cannot prove either case, we fail, which is inconclusive.

Notice our correctness proof above only requires that $\text{PROVE-CONSISTENT}(t, k)$ does not erroneously return true. An algorithm that always fails satisfies this correctness criteria, but is obviously not useful. We show that our algorithm is useful by implementing it, as described in Section V, and using it to verify a memory-management program described in Section VI.

V. IMPLEMENTATION FOR x86 IN ACL2

We added a TLB and implemented our reasoning strategy for the ACL2 model of the x86 ISA [2]. We chose ACL2 [1] and the aforementioned x86 model for several reasons: The x86-ISA model supported and had a mature theory for reasoning about address translation by walking page tables. ACL2’s capacity for handling large proof terms and the high degree of automation facilitated by ACL2’s primary proof procedure, its rewriter, makes it well suited for working with the large terms that arise from the symbolic execution of programs. The efficient executability of the ACL2 logic makes it possible to validate and debug the ISA model through testing.

Our ACL2-based model of the x86 ISA is written as an interpreter of x86 machine code. At its core, is a function that, given a machine state, decodes the instruction pointed to by the current instruction pointer, and then executes it by returning an updated machine state as specified by the x86-ISA documentation [12]. This function can be invoked repeatedly, producing an interpreter that executes a sequence of instructions. Since the interpreter is a function in the ACL2 logic, we can use the ACL2 theorem prover to prove theorems about its behavior. It is also a program written in the ACL2 programming language, so in addition to being a formal specification, the interpreter can be executed. It has support for about 500 x86 instructions, supports exceptions, interrupts, and address translation and is complete enough to boot Linux and execute the GNU C compiler as a Linux process. The executability of the model makes it possible to run tests to validate the accuracy of our ACL2 x86-ISA model and use cosimulation, executing our interpreter instruction by instruction in lock step with a trusted x86 implementation, to find bugs in our model.

The ACL2 theorem prover’s primary proof procedure is its rewriter that applies conditional rewrite rules to simplify terms. This rewriter can be programmed by an ACL2 user by proving theorems which are stored in the rewriter’s database as rewrite rules. When the rewriter finds a term that matches the left-hand side of a rewrite rule, it recursively attempts to rewrite the hypotheses of the rule to true; if it succeeds, the matched term is rewritten as specified by the rewrite rule. This allows a skilled ACL2 user to develop rewrite rules that can automatically be applied to simplify large proof terms without

user intervention, greatly automating the proof process. This, along with the rewriter’s ability to manipulate large proof terms efficiently, has made ACL2 a good choice for proofs involving large terms, like those that arise when executing programs symbolically. This is one reason why ACL2 has found use in industry for the verification of hardware designs at corporations like Centaur Technology [13] and Intel.

Goel developed a library of theorems to aid with the verification of x86 software using the x86-ISA model. These included many lemmas about address translation behavior that Goel used to verify a memory-management program called Zero-Copy [14]. This work was done on a model lacking a TLB, but because our approach reduces address translation in the presence of a TLB to address translation by walking the page tables, we were able to reuse much of Goel’s work and re-verify an updated version of the Zero-Copy program on the ACL2, x86-ISA model with a TLB. We discuss this further in Section VI.

A. TLB Implementation

Our TLB is modeled as a mapping from virtual-page numbers, various control bits that control address translation, and the access type (i.e., read, write, or execute) to physical-page numbers. Our TLB treats all memory addresses as being contained within 4KB pages and may use multiple entries corresponding to naturally aligned 4KB regions to cache translations for larger page sizes. The x86 ISA requires implementations to invalidate one or more TLB entries on certain events. For example, when executing the `INVLPG` instruction all TLB entries corresponding to the page containing the given virtual address must be invalidated. Whenever an event occurs that requires one or more TLB entries to be invalidated, our model clears the TLB in its entirety. While our choice to cache large pages with multiple 4KB region entries and invalidate the entire TLB whenever we are required to invalidate any TLB entry may seem odd, our implementation is allowed by the Intel x86 Software Developer’s Manual [12].

This design was chosen in part to maximize the speedup it gave in the execution performance of our x86-ISA model because when executing x86 programs using the model as an interpreter, a significant portion of the runtime is spent translating addresses. With this design, we can represent TLB mappings using the ACL2 *fast-alist* mechanism [15]. In the ACL2 logic, a fast-alist is an association list (or alist), a list of key-value pairs, but in execution, ACL2 associates with the alist a hash-table. To speed up execution, ACL2 maintains the invariant that looking up a key in the hash-table yields the same value as the value associated with the key in the alist. This allows us to have hash-table performance while retaining the reasoning simplicity of a list of key-value pairs, but prevents us from removing key-value pairs from the alist. Thus, we choose to clear the TLB by replacing the TLB alist with an empty one whenever we need to invalidate entries.

When our x86-ISA model translates a virtual memory address to a physical address, it takes the virtual-page number of the address being translated, the current state of the processor

control bits that affect address translation, and the memory-access type, and it searches for a corresponding virtual-page number in our TLB. If it finds an entry, it uses the corresponding physical-page number to create a translated address by concatenating the lower 12 bits from the virtual address to the physical-page number found in the TLB. If no entry corresponding to the translation is found in the TLB, our x86-ISA model performs a page-table walk to translate the virtual address. If this translation fails, it triggers a page fault and does not update the TLB. Otherwise, it updates our TLB by adding an entry corresponding to the translation that maps to the physical-page number returned by the page table walk. Unlike a hardware TLB, our TLB model can grow to contain an arbitrary number of entries.

B. Reasoning Implementation

We implement PROVE-CONSISTENT implicitly by proving theorems that characterize the maintained and consistent set of translations for each operation of our x86 model. Then, when we attempt to prove a theorem that states a translation is TLB consistent after a sequence of operations, the ACL2 rewriter will attempt to apply our rewrite rules in a manner that implements the PROVE-CONSISTENT algorithm. If the sequence of operations is empty, it will attempt to prove the translation is consistent with respect to the initial state. If the translation under consideration can be proven to be in the set of maintained translations of the outer-most operation the ACL2 system, due to the rewrite rule associated with the theorem that describes the set of maintained translations of the outer-most operation, will rewrite the conjecture to prove that the address translation is TLB consistent with respect to the previous state. If the translation can be proven to be in the set of consistent translations of the outer-most operation, it will confirm the validity of the conjecture. This is exactly the PROVE-CONSISTENT algorithm.

We defined TLB consistency for the x86-ISA model in ACL2, shown in Figure 3. Recall, we informally defined TLB consistency to mean the TLB does not contain any stale entries corresponding to a given translation. Our formal definition checks that performing the given translation by first consulting the TLB, and falling back on the page tables if there is no relevant TLB entry, yields the same result as performing the translation by walking the page tables directly, without consulting the TLB. Since a stale TLB entry is by definition a TLB entry that is inconsistent with the page tables, our formal definition of TLB consistency captures our informal definition.

We define rewrite rules that describe the set of maintained translations and the set of consistent translations of the operations of the machine. For example, Figure 4 shows a theorem that states that all translations are maintained by the address translation operation. Since this theorem has no hypotheses it is stored as an unconditional rewrite rule by the ACL2 theorem prover. This rule programs the rewriter to rewrite terms that are calls to `tlb-consistent` composed with the address translation operation. Such terms are rewritten to calls of `tlb-consistent` on the state the address translation

operation was applied to, i.e., the state before the address translation operation was applied. It is also possible for us to define the set of maintained operations in terms of the parameters to the operation or the machine state, as shown in Figure 5; this theorem states that a translation remains TLB consistent after a write to physical memory if that write does not change any page table entry used when performing the translation. This lemma is stored as a conditional rewrite rule in the ACL2 prover; when the ACL2 rewriter matches a term to the left-hand side of the equality, it attempts to recursively rewrite the hypotheses to true. If it succeeds, it will rewrite the term to the right-hand side.

In our x86-ISA model, we have only a single operation that has a non-empty set of consistent translations because clearing the TLB is the only way we invalidate TLB entries. Figure 6 shows a theorem which states that a machine state with empty TLB is consistent for all translations. While we could have instead proven all translations are consistent after the clear TLB operation, this theorem is more general, and the rewriter will easily be able to apply it in the same way when proving a translation is TLB consistent after the clear TLB operation. When the rewriter sees any `tlb-consistent` term, it will attempt to prove that the TLB in the given machine state is empty. If it succeeds, it will rewrite the `tlb-consistent` term to true.

VI. CASE STUDY: ZERO-COPY

We demonstrate the effectiveness of our TLB reasoning technique by proving the correctness of a supervisor memory-management program. Zero-Copy is a program that copies a page of data in the virtual-address space to another page in the virtual-address space. It does this not by copying the data but instead by modifying the page tables so that the destination virtual addresses translate to the same physical addresses as the source virtual addresses. Goel proved the correctness of this program on the x86-ISA model without a TLB [14]. Since Goel’s program was written for a version of the x86-ISA model that lacked a TLB, it did not attempt to invalidate TLB entries. Under certain assumptions, Goel proved the following theorems to verify Zero-Copy:

Theorem 1 (Source Unmodified): After the program runs, the data in the source region is identical to the data in the source region prior to the run.

Theorem 2 (Destination Updated): After the program runs, the data in the destination region is the same as the data in the source region before the program runs.

Theorem 3 (Program Unmodified): After the program runs, the program in memory is the same as the program in memory before the program runs.

We added code to Zero-Copy that invokes the x86 `INVLPG` instruction to invalidate TLB entries in the destination region after updating the page tables to avoid the TLB containing stale entries, which is necessary for the correctness of the program. The only TLB-related change we made to the statements of Goel’s correctness theorems was the addition of hypotheses that required translations used by the program to be

```

(define tlb-consistent ((lin-addr canonical-address-p)
  (r-w-x      :type (member :r :w :x))
  x86)
...
(b* ((lin-addr (mbe :logic (logext 48 (loghead 48 lin-addr))
  :exec lin-addr))
  ((mv flg phys-addr &) (ia32e-la-to-pa lin-addr r-w-x x86))
  ((mv flg2 phys-addr2 &) (ia32e-la-to-pa-without-tlb lin-addr r-w-x x86)))
  (and (equal flg flg2)
    (equal phys-addr
      phys-addr2)))
...))

```

Fig. 3. Function `tlb-consistent` is the ACL2 definition of TLB consistency for our x86-ISA model. `ia32e-la-to-pa` performs an address translation by consulting the TLB first. If it finds a relevant TLB entry, it uses the entry to perform the translation. If no relevant TLB entry was found, it uses the page tables to perform the translation. `ia32e-la-to-pa-without-tlb` performs an address translation using only the page tables. `tlb-consistent` returns true if and only if the given translation has the same result when translated with both of the aforementioned functions. Some details not relevant to the logical definition have been elided.

```

(defthm translating-addresses-maintains-consistency
  (equal (tlb-consistent lin-addr r-w-x
    (mv-nth 2 (ia32e-la-to-pa lin-addr2 r-w-x2 x86)))
    (tlb-consistent lin-addr r-w-x x86))
  ...)

```

Fig. 4. ACL2 definition of `translating-addresses-maintains-consistency`, a theorem that states a translation is TLB consistent after the address translation operation if and only if it was TLB consistent before the address translation operation. This is a stronger statement than the address translation operation's set of maintained translations is all translations because it is biconditional instead of only an implication. This theorem is stored as a rewrite rule that can be used by the ACL2 rewriter to rewrite appropriate terms. Hints to the prover to help it prove this theorem have been elided.

```

(defthm writing-non-page-table-memory-maintains-tlb-consistency
  (implies (disjoint-p (list index)
    (xlation-governing-entries-paddrs (logext 48 lin-addr) x86))
    (equal (tlb-consistent lin-addr r-w-x (xw :mem index val x86))
      (tlb-consistent lin-addr r-w-x x86)))
  ...)

```

Fig. 5. ACL2 definition of `writing-non-page-table-memory-maintains-tlb-consistency`, a theorem that states that writing to physical memory does not change the consistency of a translation if the write is to an address that is not part of a page table entry that may be consulted when performing the translation. Like the theorem in Figure 4, it is stronger than showing that the translation is in the set of maintained translations of the physical memory write operation since it states that translation's consistency is unchanged by the memory write, rather than stating that the translation is consistent after the write if it was consistent before the write. Hints to the prover to help prove this theorem have been elided.

```

(defthm empty-tlb-is-consistent
  (implies (atom (xr :tlb nil x86))
    (tlb-consistent lin-addr r-w-x x86))
  ...)

```

Fig. 6. ACL2 definition of `empty-tlb-is-consistent`, a theorem that states when the TLB is empty all translations are consistent.

TLB consistent in the machine's initial state. The correctness theorems do not hold without these hypotheses. We also added minor hypotheses unrelated to the TLB to accomodate other changes made to the x86-ISA model between when Goel initially verified the Zero-Copy program and when we verified the updated Zero-Copy program with our TLB enhanced version of the x86-ISA model.

Using our methodology, we were able to reverify Zero-Copy without significant changes to the structure of the proof. Our approach involves showing that the translations the

program uses are TLB consistent and then using that fact to reason about the results of address translation as though the translation was performed using the page tables. This allowed us to avoid having to make major modifications to Goel's proofs since, when a translation is TLB consistent, the results of the translation are the same as they would have been in the model Goel used that lacked a TLB.

Since we implemented our reasoning strategy using rewrite rules, we were able to exploit ACL2's rewriter and reprove many theorems automatically. While most of the lemmas about

address translations and virtual memory that Goel used in her proof were not theorems in the updated x86-ISA model with a TLB, they were theorems after adding hypothesis that require the translations relevant to the lemma to be TLB consistent. After adding such hypotheses to the various lemmas in Goel’s proof, they could often be re-proven automatically. Other theorems required additional theorem-prover hints. Many of these updates were done to the library Goel developed for x86-ISA, machine-code verification, and these updated lemmas can be used directly to verify other programs.

The most involved changes were in the proof of a lemma that stated if a leaf 1 GB page table entry is updated and then values are read from the virtual-memory region corresponding to the page table entry, the values read will be equal to the values in the page at the physical address specified in the updated page table entry. After adding the condition that TLB consistency must hold for all the read translations in the virtual-memory region after the page table entry is written, the proof of this particular lemma was redone with significant changes to how it was structured. In principle, with the old line of reasoning it should have been possible to update using our method, but ACL2 could not automatically re-prove the lemmas, and we decided to redo much of the proof to make it easier to understand and maintain.

Using our approach minimized the complexity of reasoning with the x86-ISA model that was introduced by the TLB’s semantics. Proofs that concern code that does not update the page tables was largely done in the same fashion as proofs done on the model without a TLB because our technique allowed the ACL2 rewriter to often automatically discharge TLB-consistency hypotheses, which were the primary complication introduced by the TLB when reasoning about such code. Proofs regarding code that updates the page tables required more involved changes in reasoning, but these changes were localized to only the small pieces of reasoning concerning page table updates. Future work could abstract some of these pieces of reasoning into a library that can be shared between proofs to minimize the amount of work needed for reasoning about page table updating code.

VII. CONCLUSION & FUTURE WORK

We described a procedure to reduce theorems about address translation in the presence of a TLB cache to address translation without caching. We added a TLB to the ACL2 x86-ISA model and implemented our algorithm implicitly using ACL2 rewrite rules. Our reduction is performed by proving that the translations we are considering are TLB consistent, meaning if there is a relevant entry in the TLB it agrees with the page tables. We then successfully applied our methodology to re-verify the Zero-Copy program that had been verified by Goel with a previous version of the x86-ISA model that did not have a TLB. We found that our methodology allowed us to reuse much of her proof. We have released our implementation and proofs under the 3-clause BSD license, and they are available, along with documentation [4], in the ACL2 Community Books [3].

There is much more to be done with respect to reasoning about memory-management software for contemporary microprocessors. To comprehensively verify security and correctness of operating systems, it will be necessary for the formal-methods community to continue the formalization of the instruction set architectures that are employed by contemporary computer systems. In particular, we would like to see address translation cache models extended to other address translation caches beyond TLBs, like page table entry caches. Additionally, we would like to see our reasoning technique implemented for a TLB model which does not make any assumptions about the TLB microarchitecture beyond that it is consistent with the ISA.

ACKNOWLEDGMENT

The authors would like to thank the ACL2 developers and community for their help over the 15+ years we have been pursuing this project. This work was supported, in part, by Intel Corporation.

REFERENCES

- [1] M. Kaufmann and J. S. Moore, “ACL2,” <https://www.cs.utexas.edu/users/moore/acl2/>, 2025.
- [2] M. K. Shilpi Goel, Warren A. Hunt Jr., “Engineering a formal, executable x86 isa simulator for software verification,” in *Provably Correct Systems (ProCoS)*. Springer International Publishing, 2017, pp. 173–209.
- [3] The ACL2 Community, *The ACL2 Community Books*. <https://github.com/acl2/acl2/tree/master/books>, 2024.
- [4] —. (Accessed, August 2025) X86isa. [Online]. Available: https://www.cs.utexas.edu/~moore/acl2/manuals/latest/?topic=ACL2_X86ISA
- [5] ACL2, *User manual for the ACL2 Theorem Prover and the ACL2 Community Books*, accessed 2025-08-13. [Online]. Available: <https://www.cs.utexas.edu/~moore/acl2/manuals/latest>
- [6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “sel4: formal verification of an os kernel,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: <https://doi.org/10.1145/1629575.1629596>
- [7] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, “Comprehensive formal verification of an os microkernel,” *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb. 2014. [Online]. Available: <https://doi.org/10.1145/2560537>
- [8] H. Syeda and G. Klein, “Reasoning about translation lookaside buffers,” in *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPIc Series in Computing, T. Eiter and D. Sands, Eds., vol. 46. EasyChair, 2017, pp. 490–508. [Online]. Available: <https://doi.org/10.29007/c2f1>
- [9] —, *Program Verification in the Presence of Cached Address Translation*. Springer, Lecture Notes in Computer Science, July 2018, pp. 542–559.
- [10] A. Fox and M. O. Myreen, “A trustworthy monadic formalization of the armv7 instruction set architecture,” in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–258.
- [11] ARM Corporation. (Accessed, May 2025) Arm architecture reference manual armv7-a and armv7-r edition. [Online]. Available: <https://developer.arm.com/documentation/ddi0406/latest/>
- [12] Intel Corporation, *Intel 64 and IA-32 Architecture Developer Manuals*. Intel Corporation, 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [13] W. A. Hunt Jr., 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.

- [14] S. Goel, *Formal Verification of Application and System Programs based on a Validated x86 ISA Model*. The University of Texas, 2016. [Online]. Available: <https://repositories.lib.utexas.edu/server/api/core/bitstreams/858b2f9b-5532-4b2a-bed1-fb889a265f6c/content>
- [15] B. Boyer and W. A. H. Jr. (Accessed, May 2025) Fastalists. [Online]. Available: https://www.cs.utexas.edu/~moore/acl2/v8-5/manual/index.html?topic=ACL2____FAST-ALISTS

Making Rabbit Run for Security Verification of Networked Systems with Unbounded Loops

Sewon Park 
 Graduate School of Informatics
 Kyoto University
 Kyoto, Japan
 sewon@sewonpark.com

Atsushi Igarashi 
 Graduate School of Informatics
 Kyoto University
 Kyoto, Japan
 igarashi@kuis.kyoto-u.ac.jp



Abstract—Rabbit is a modeling language for networked systems that allows processes and their communications to be programmed using familiar imperative constructs, while enabling various security assertions to be automatically verified via the Tamarin prover as a back-end. We propose an extension of Rabbit with general unbounded and nondeterministic loops, which are essential for modeling more realistic scenarios, along with optimization techniques that keep automatic verification via Tamarin tractable. We further introduce parameterized processes and communication channels, making Rabbit expressive enough to model systems such as a centralized server communicating with an unbounded number of client processes via a family of parameterized channels. We provide a proof-of-concept implementation of the extended Rabbit and present evaluation results.

Index Terms—Security Verification, Networked Systems, Modeling Language, Tamarin Prover, Formal Verification

I. INTRODUCTION

Verifying networked systems has been a central goal in cybersecurity as more sensitive information is being communicated through open networks. Inaba et al. [1] have developed a modeling language called Rabbit to help system developers (who are not necessarily experts in formal verification) formally specify the target system from a security point of view, namely, what are assets to be protected, how various kinds of data flow in the target system, what are attack surfaces, and so on. Although its language design was partially inspired by other modeling languages such as UMLsec [2], [3], Rabbit features low-level notions such as processes, file systems, memory, and network communications and provides an imperative language to describe data flow and an access control mechanism for flexible attacker modeling. It is implemented as a translator to a model of the Tamarin prover [4], a model checker for automatic verification, and has been used to model and verify a system with a centralized server that validates signed camera images sent sequentially by clients.

Our work begins by filling a gap and proposes a significant extension to Rabbit. In [1], although the language is designed to resemble familiar programming styles, general loops were omitted for performance reasons. Only bounded loops are supported, which are unrolled at translation time. However, to model realistic networked systems, it is desirable to support loops, possibly unbounded, that repeatedly make transactions

without a predetermined limit. In popular security verification tools such as SAPIC+ [5] or ProVerif [6], which are based on the applied π -calculus, an obvious choice is to adopt the replication construct $!P$ to express unbounded loops. We prefer having loops as a primitive because not only is it natural for ordinary system developers but also it allows loop-specific optimizations, as we discuss later in Section V.

We replace an ad-hoc construct to express remote procedure call handlers in [1] with a variant of the guarded loop/conditional commands found in Dijkstra’s Guarded Command Language [7]. We adopt *facts*, predicates in the global system state that serve as the basic means of inter- and intra-process communication in multiset rewriting models of protocols [8], to act as guards so that the guards can represent, for example, messages received through communication channels. Our guarded commands are general enough to express sequential loops, non-deterministic reactive processes (such as remote procedure call handlers), and file systems.

Our novelty in adopting facts is introducing their scoping. A fact denoting a state in a channel is scoped within the channel entity, and a fact denoting a local state (which we call a structure-fact, handled only indirectly by structure commands) is scoped within the process and intended to be effect-free to the external environment of the process. The latter concept of process-locality enables local optimization, which we deem necessary to make automatic verification of our extension tractable; cf. SAPIC+ [5, Section E.2], which provides a similar optimization but not process-local facts.

Another important aspect missing in [1] is parameterization. There, only a fixed and finite number of processes can be declared. We further extend Rabbit with parameterized processes and support for unbounded instantiation of those processes.

Our main contribution in this paper is extending the command language of Rabbit, introducing nondeterministic cases and loops, formalizing their semantics, and presenting optimization techniques to make them feasible in practice, overcoming the limitations identified in [1]. Other important aspects of Rabbit, such as access control policies, are not addressed here.

We organize the paper as follows. In Section II, we define the syntax of the extended Rabbit, focusing on the newly introduced constructs and explaining their intended meanings.

(Unless stated otherwise, Rabbit refers to this extended version.) In Section III, we define its formal semantics in terms of transition graphs and introduce an optimization technique, graph compression, which abstracts some local interactions. In Section IV, we introduce parameterization, and in Section V, we present the implementation of Rabbit and its translation to Tamarin. There, we also discuss another optimization technique for Rabbit’s loops, which we argue is necessary to make verification of loops tractable. In Section VI, we present an advanced example system adapted from [1], now featuring unbounded loops and unbounded processes. We compare the verification time with other leading software, SAPIC+ [9], [5] and ProVerif [6], to justify our choice of providing loops as a primitive supported by loop- and Rabbit-specific optimizations. In Section VII, we conclude the paper by listing some directions for future work.

II. SYNTAX

A system model in Rabbit consists of an equational theory (which is an algebraic specification of security primitives), global function definitions, an access control policy (which also serves as the basis for a *configurable* Dolev–Yao symbolic attacker model [10]), an initial configuration of resources (such as channels and file systems), process definitions with their local functions, and security assertions to be verified. We focus on elements related to our extension and refer readers to [1] for the details of other elements.

A. Expressions

Each Rabbit system is defined over an *equational theory*, which determines the expression language of the system, similarly to [11] and other frameworks. For example, before defining a system, we may declare

```
function pub:1 aenc:2 adec:2
equation adec(aenc(x, k), pub(k)) = x
```

for public key generation, asymmetric encryption, and decryption operations: function symbols `pub` of arity 1, and `aenc` and `adec` of arity 2, along with an equation that characterizes their behavior. Once defined, these operations can be used in the expression language, for example in an assignment like $y := \text{aenc}(x, k)$, where y is a programming variable.

Besides any declared equational theory, Rabbit expressions e consist of lists $\langle e_1, \dots, e_n \rangle$, the unit $\#$, string literals “ ℓ ”, programming variables, and logical variables. Programming variables are mutable and reside within each process, while logical variables are used to capture values.

B. Facts

All programming variables in Rabbit are local to individual processes. Therefore, using only variables, we cannot describe the state of a channel – what is stored, requested, et cetera. Instead, we use various kinds of *facts* [4], which are syntactically just formal terms, to describe the global state of a system. For an expression e that represents a channel, a **channel fact** $e::\text{Store}(v)$ stating that $\text{Store}(v)$ holds in e may be used to model that v is “stored” in e . Our notation for channel facts

is motivated by namespace scoping mechanisms in languages such as C++. (The fact name `Store` should be chosen by the user and has no predefined meaning. Its meaning depends entirely on how the user program treats it.)

Given two processes both connected to a channel `udp`, one process may place the channel fact `udp::Store( $e$ )` into the environment to upload the value e to the channel. The other process can then retrieve e by checking for the presence of the fact `udp::Store( $x$ )` for some x . This is how inter-process communication is realized in Rabbit.

There are two **attacker facts**: `Out( $e$ )` denoting a global state where the attacker hears the value e and `In( $e$ )` denoting a global state where the attacker tries to feed e . They are meant to match those in Tamarin and model attacks in Rabbit. For example, an operation having a side-effect of putting `Out( $e$ )` to the environment models leaking e and an operation returning e from `In( $e$ )` models an attack where the attacker contaminates the process’s operation with its knowledge e .

There are two obvious **equational facts** $e_1 = e_2$ for equalities and $e_1 \neq e_2$ for inequalities of expressions, up to the based equational theory.

In each process, for an expression e that represents a path in its file system, the **file fact** $e.d$ for an expression d , denotes the state of the file system where d is stored in e .

Note that some facts are local (file facts are local to a process), while others are global (attacker facts, equational facts, and channel facts). Most facts are linear, in the sense that they can be introduced to restrict the system state (for example, to indicate that a channel holds a message) or consumed to relax it (meaning the channel no longer needs to store that message). In contrast, equational facts are persistent.

C. Tags

Tags do not have any computational effect but are used to mark moments in the computation so they can be referred to later in assertions. A tag may take arguments. For example, in the tag $T(e_1, \dots, e_n)$, the expressions $e_1, \dots, e_n$ can later be referenced by their values at the moment the tag was applied. There is one reserved **attacker tag**, `K( $e$ )`, corresponding to an event where the attacker learns the value of e .

D. Commands

The full syntax of our command language is found in Figure 1. There, $\vec{A}$ refers to an arbitrary list of facts, T refers to a tag, and S refers to a structure name, whose use will be explained later.

The command language is presented in three blocks. The first consists of familiar constructs found in imperative programming languages. The function f in a function call refers either to a local function defined within the process that executes the command or to a globally defined function. Global functions can be used to model system calls shared among processes. We design commands to subsume expressions and treat variables as local, which simplifies the definition of functions: a function can be defined as a command, without

$c ::= e$	expressions
skip	doing nothing
var $x := e$ in c	local variable (x mutable in c)
$x := e$	assignment
$c_1; c_2$	sequencing
$x := f(e_1, \dots, e_n)$	function call
put $[\vec{A}]$	output
case $[\vec{A}_1] \Rightarrow c_1 \mid \dots \mid [\vec{A}_n] \Rightarrow c_n$	guarded cases
end	
repeat $[\vec{A}_1] \Rightarrow c_1 \mid \dots \mid [\vec{A}_n] \Rightarrow c_n$	guarded loop
until $[\vec{A}'_1] \Rightarrow c'_1 \mid \dots \mid [\vec{A}'_m] \Rightarrow c'_m$	
end	
event $[T]$	tag
new $x := S(e_1, \dots, e_n)$ in c	allocation
let $x_1, \dots, x_n := e.S$ in c	fetch
delete $e.S$	deletion

Fig. 1. Command language of the extended Rabbit

requiring an explicit return. The return value of a function is then given by the value of the command itself.

The second block is Rabbit-specific. The output command inserts the linear facts $\vec{A}$ into the environment, constituting the only mechanism in Rabbit for outputting a message to a communication channel. Of course, it is not limited to channels; it can also be used to create a file, e.g., by put $[e_1.e_2]$.

The guarded cases execute as follows: when for some i , $\vec{A}_i$ holds in the fact environment, it consumes the matching linear facts and runs the corresponding guarded command c_i . If multiple branches are valid, one is chosen nondeterministically.

The guarded loop repeatedly performs the guarded cases while testing the exit conditions $\vec{A}'_i$. When one of the exit conditions holds, say $\vec{A}'_i$, it may consume linear facts of $\vec{A}'_i$, execute c'_i , and then exit the loop. It is nondeterministic in the sense that the loop may continue even when an exit condition holds, as long as some iterating condition remains satisfied.

The guarded command $[\vec{A}] \Rightarrow c$ provides a mechanism in Rabbit for reading a message from a communication channel. The facts in $\vec{A}$ may contain free variables that appear in c . These are local logical variables bound by the pattern in $\vec{A}$. For example, in $[\text{udp} :: \text{Store}(x)] \Rightarrow c$, where c contains a free logical variable x and the environment contains $\text{udp} :: \text{Store}(e)$ for some expression e , the command c is executed with x bound to e . As linear conditions are consumed, the fact $\text{udp} :: \text{Store}(e)$ is removed from the environment.

The tag command annotates the current point of computation with a tag T . This tag is referenced later in assertions but has no effect on the computation itself.

The final block introduces structure-related constructs, refer-

ring to struct-like data structures in C, data structures with a finite list of (unnamed) fields. We propose using structures as a restricted mechanism to represent local state within processes. The allocation command creates a structure S with arguments $e_1, \dots, e_n$, allocates fresh memory for it within the process, and binds a new local logical variable x within the command c to the resulting address. However, unlike the variable x , the allocated structure and the bound address may escape c .

The fetch command, when e refers to an address holding a structure S , retrieves its components $e_1, \dots, e_n$ and binds the local logical variables $x_1, \dots, x_n$ within c to the components.

The deletion command, when e refers to an address containing a structure S , deletes the structure at that address.

When a structure name is omitted, as in new x in c , a trivial structure is created, and x denotes a fresh nonce within c , serving as the nonce generation mechanism in Rabbit.

E. Examples

1) *Unbounded Loops*: We can express two types of unbounded loops. The first allows nondeterministic repetition:

```
repeat []  $\Rightarrow$   $c_1$  until []  $\Rightarrow$   $c_2$  end
```

The second expresses indefinite repetition:

```
repeat []  $\Rightarrow$   $c_1$  until ["0" = "1"]  $\Rightarrow$  skip end
```

Note that their intended meanings differ. In the first case, there may exist both terminating and non-terminating executions. In contrast, the second always results in an infinite loop.

2) *Remote Procedure Call (RPC)*: RPC calls were provided as primitives in the original Rabbit, whereas we can now program an RPC handler using the new constructs, making them more closely resemble the behavior of realistic systems.

Suppose a process defines local functions $f_1, \dots, f_n$ and associates each with an integer identifier $1, \dots, n$. Then, we can program the RPC handler as follows:

```
repeat [ch::invoke( $f, x$ )]  $\Rightarrow$ 
  case [  $f = \text{"1"}$  ]  $\Rightarrow$ 
    var  $r := f_1(x)$  in put [ch::return( $r$ )]
  | ... | [  $f = \text{"n"}$  ]  $\Rightarrow$ 
    var  $r := f_n(x)$  in put [ch::return( $r$ )]
  | [  $f \neq \text{"1"}, \dots, f \neq \text{"n"}$  ]  $\Rightarrow$ 
    event [InvalidFunction( $f$ )]
  end
until [ch::exit()]  $\Rightarrow$  event[ExitRPC()]
end
```

The caller can put $\text{ch} :: \text{invoke}(f, x)$ to invoke the function $f(x)$ on the callee and listen for the response via $\text{ch} :: \text{return}(r)$, or send $\text{ch} :: \text{exit}()$ to terminate the connection. If f is not a valid function on the callee side, the handler emits the tag $\text{InvalidFunction}(f)$.

3) *File System*: The structure commands are used to model local system calls. Suppose we have an expression p representing a path in the file system. Then, the system call for opening the file at p can be expressed as follows:

```
case [ $p.y$ ]  $\Rightarrow$ 
  put [ $p.y$ ]; new  $fd := \text{FD}(y)$  in  $x := fd$  end
```

This command, assuming that p is a valid path for a file storing y , creates a fresh value fd as a file descriptor holding the structure $FD(y)$ and assigns it to a mutable variable x . Note that we put the file fact back into the environment; without it, $p.y$ gets dismissed after the command, which would correspond to deletion of the file.

Later, we can read the file content from the descriptor using:

```
let y := x.FD in c
```

which binds the logical variable y in c to the file content. To close the opened file, we can delete the structure:

```
delete x.FD
```

F. Process, System, and Assertions

A **process** P consists of communication channels, files with stored data (as expression pairs), top-level mutable variables, local functions (with arguments and commands), and a main command governing execution. A process has access only to the channels given to it. Any attempt to access a channel that is not given to the process, such as using $[udp::Store(x)] \Rightarrow c$ when udp is not available to the process, will be forbidden at the semantic level.

A **system** is a finite list of processes. (Later in Section IV, we consider systems with an unbounded number of running processes.)

Although the full assertion language of Tamarin is supported, this paper focuses on two specific types of assertions:

- The **reachability assertion**

Reachable $\vec{T}$

states that there is an execution in the system where all tags in $\vec{T}$ occur.

- The (non-injective) **correspondence assertion**

Corresponds $T_1 \rightsquigarrow T_2$

between two tags T_1 and T_2 states that for any execution in the system where T_1 occurs, there must have been a corresponding moment when T_2 occurred in the past.

Reachability is useful for expressing properties like liveness of the system or secrecy of an expression, while correspondence is useful for expressing authenticity; see [12].

III. SEMANTICS

A. Transition Graph

Commands in Rabbit evaluate to values, and operations such as $\text{var } x := e \text{ in } c$ require a register in their realization that carries the value of c as the value of the entire command. To support this, we introduce an expression construct ρ , a symbol representing the value stored in the register.

We assume standard well-formedness judgments for expressions, commands, facts, tags, and processes written as $\Gamma \vdash e$, $\Gamma \vdash c$, $\Gamma \vdash \vec{A}$, $\Gamma \vdash T$ and $\vdash P$, respectively. The context Γ is a pair $(\Gamma_l; \Gamma_m)$ of distinct lists of logical variables Γ_l and of mutable variables Γ_m . We also assume a fixed environment of function definitions, which we omit from the notation.

We write $\cdot$ to denote the empty list. It holds that $(\cdot; \cdot) \vdash \rho$.

For a well-formed process P , to define its operational semantics, we construct a directed graph G_P . The **transition graph** consists of vertices corresponding to the program locations in the process's definition. We use a multi-**index**, a non-empty list $\vec{i}$ of pairs of integers, to denote a vertex, which includes the index and its associated context $\Gamma^{\vec{i}}$. Since each index identifies at most one vertex, it is safe to refer to a vertex by its index. We define the following index operations: $\mathbf{0} := (0, 0)$, $((n, i) : \vec{i}) + k := (n, i + k) : \vec{i}$, and $\vec{i}_n := (n, 0) : \vec{i}$ where $:$ is the cons operator.

An edge $\tau : \vec{i} \rightarrow \vec{j}$ is labelled as

$$[\vec{A}]_{\Gamma^{\vec{i}}} \xrightarrow{f, [T]} [\vec{B}]_{\Gamma^{\vec{j}}}$$

representing a small-step transition. The **precondition** $\vec{A}$ and **postcondition** $\vec{B}$ are lists of facts extended with additional kinds: (1) **freshness facts** $\text{Fr}(e)$, meaning that e is a fresh value; and (2) **structure facts** $S(\kappa; e, e_1, \dots, e_n)$ for each structure S , meaning that in the process represented by κ , at address e , the structure $S(e_1, \dots, e_n)$ is stored.

The tag T is a tag extended with (1) a loop start indicator $\text{Loop}_{\text{in}}(\kappa, \vec{i})$, (2) a looping indicator $\text{Loop}_{\text{back}}(\kappa, \vec{i})$, and (3) a loop exit indicator $\text{Loop}_{\text{out}}(\kappa, \vec{i})$, indicating the location $\vec{i}$ of loop-related events in the process represented by κ . Here, $\vec{i}$ denotes the string representation of $\vec{i}$.

The conditions and the tag are well-formed in $\Delta, \Gamma_l^{\vec{i}}; \Gamma_m^{\vec{i}}$ for some list Δ of fresh logical variables such that $\Gamma_l^{\vec{i}}$ is a sublist of $\Delta, \Gamma_l^{\vec{i}}$, denoted by $\Gamma_l^{\vec{j}} \subset \Delta, \Gamma_l^{\vec{i}}$.

The case where the precondition refers to a fresh logical variable that is not in $\Gamma_l^{\vec{i}}$ but appears in $\Gamma_l^{\vec{j}}$ arises when a new logical variable is introduced via the pattern $\vec{A}$ in a guarded command $[\vec{A}] \Rightarrow c$. For example, the variable x in $[udp::Store(x)] \Rightarrow c$ may not be in the initial context when encountering $udp::Store(x)$, but it will be present in the context executing c , having been introduced by the guard.

For a list of variables Δ , define $\llbracket \Delta \rrbracket$, the set of valuations, recursively as $\llbracket \cdot \rrbracket = \{\cdot\}$ and $\llbracket x, \Delta \rrbracket = \text{Val} \times \llbracket \Delta \rrbracket$ where Val is the set of values, a set of variable-free expressions. For a context Γ , we write $\llbracket \Gamma \rrbracket$ for the set of **local states** in Γ :

$$\llbracket \Gamma \rrbracket := \text{Val} \times \llbracket \Gamma_l \rrbracket \times \llbracket \Gamma_m \rrbracket$$

where the first component expresses the value at the register. Given a well-formed expression $\Gamma \vdash e$, its evaluation is a mapping of form $\llbracket \Gamma \vdash e \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \text{Val}$ where it holds that $\llbracket \Gamma \vdash \rho \rrbracket(x, \gamma_l, \gamma_m) = x$ for all γ_l and γ_m .

The **state update** f , when $\Gamma_m^{\vec{j}} = x_1, \dots, x_n$, is a pair $(f_\rho; f_{x_1}, \dots, f_{x_n})$ of an expression f_ρ for the register and expressions $f_{x_1}, \dots, f_{x_n}$ for mutable variables, all well-formed in $\Gamma^{\vec{i}}$. It denotes the mapping $\llbracket f \rrbracket : \llbracket \Gamma^{\vec{i}} \rrbracket \rightarrow \text{Val} \times \llbracket \Gamma_m^{\vec{j}} \rrbracket$, defined by evaluating the expressions. It specifies how the register and mutable variables are updated during the transition. (We omit the straightforward definition of $\llbracket f \rrbracket$ for brevity.)

We define $\langle \Gamma \vdash c \rangle \vec{i}$ to be a pair consisting of a transition graph G and a final index $\vec{j}$, such that all edges in G direct from $\vec{i}$ to $\vec{j}$ and $\Gamma^{\vec{i}} = \Gamma^{\vec{j}} = \Gamma$, inductively on the command

constructs. For example, the transition graph of a local variable creation is:

$$\begin{aligned} &\langle \Gamma \vdash \text{var } x := e \text{ in } c \rangle i = \\ &\text{let } (G, j) \leftarrow \langle \Gamma_L; x, \Gamma_m \vdash c \rangle (i+1) \text{ in} \\ &(\{ \{ \Gamma \xrightarrow{(t; e, \Gamma_m)} \Gamma_{L; x, \Gamma_m}^{i+1}, \{ \Gamma_L; x, \Gamma_m \xrightarrow{(\rho; \Gamma_m)} \Gamma^{j+1} \} \cup G, j+1 \}) \end{aligned}$$

Since a transition graph contains no isolated vertices, we identified it with its set of edges. We omit the transition graph construction for standard command constructs and instead highlight the new constructs in Figure 2. There, κ denotes the name of the process executing the commands, to be specified later in Section III-B.

When constructing the transition for a guarded command $[\vec{A}_j] \Rightarrow c_j$ in nondeterministic cases, we first generate an initial transition corresponding to the guard, from i to i_j (which should be regarded as i scoped with the branch's index j), where $\Gamma^{i_j} = \text{fv}(\vec{A}_j), \Gamma_L; \Gamma_m$. This step may introduce new logical variables appearing in $\vec{A}_j$ that are not bound in Γ^i . (Here, $\text{fv}(\vec{A})$ denotes a list of variables occurring in $\vec{A}$.) In this extended context, the guarded command is then processed, and at the final transition (from j to $i+1$), these newly introduced variables are removed, restoring $\Gamma^{i+1} = \Gamma^i$.

Guarded commands in a loop are processed similarly. The major differences are as follows: When entering a loop, we tag the transition with Loop_{in} , using the name k of the ambient process and the string representation $\bar{i}$ of the index as arguments. On loop back, we tag the transition with $\text{Loop}_{\text{back}}$, and on exit, with Loop_{out} , each referencing the initial entry. These tags have no effect on the semantics but will be used later in Section V-A for loop-specific optimizations.

For the main command c of P , that is well-formed under an initial context Γ , the transition graph G_P is $\langle \Gamma \vdash c \rangle 0$.

B. Traces and Assertions

We define *traces* as the semantics of a system. Instead of presenting the formal formulation, which follows straightforwardly from the semantics of commands, we sketch a high-level description here.

Given a system $\Sigma = P_1, \dots, P_n$ of well-formed processes, we assign their *names* using static information that distinguishes them; e.g., P_i 's name can be i or can be carefully specified by the user. Note that even when $P_1 = P_2$ by their definitions, they must have different names. Using these names, we consider their transition graphs and define the semantics of the system as system transitions on system states and fact environments. A **system state** γ maps each process P_i to its local configuration, consisting of its index in G_{P_i} and a local state γ_{P_i} . A **fact environment** E is a multiset of facts, where a special **attacker fact** $K(e)$ is added to the available kinds of facts to represent the attacker's knowledge of e .

The system evolves by transitions that either execute a process's internal transition or model the attacker's interaction with facts in E . In the first case, the system transition follows the process's internal transition: it consumes and produces facts in E , updates the system state by updating the process's

index and local state, and emits a tag. If the process's local transition accesses a channel fact for a channel ch in its pre- or postcondition, the transition is allowed only if the process has been given the channel ch in its definition.

In the second case, corresponding to an attack, the system transition preserves the system state γ but modifies the fact environment E according to the attacker's action, such as transforming $\text{Out}(e)$ into the attacker's knowledge $K(e)$ and adding $\text{In}(e)$ from $K(e)$, following the attacker model of Tamarin; see [13] for more details and other attacker actions. In the latter case, we tag the system transition with $K(e)$.

For a sequence of system transitions starting from γ and E , the sequence of tags is called a **trace** and $\llbracket \Sigma \rrbracket(\gamma, E)$ denotes the set of such traces. From the initial state γ_0 and the initial fact environment E_0 , which includes initial file facts, the attacker knowledge of public values, a countable supply of fresh values, and a persistent supply of equalities and inequalities, $\llbracket \Sigma \rrbracket(\gamma_0, E_0)$ is the semantics of the system. The validity of assertions is defined over this set of traces.

C. Graph Compression

In a transition graph, given two consecutive transitions $\tau_1 : i \rightarrow j$ and $\tau_2 : j \rightarrow k$, it is sometimes possible to safely compress them into a single transition from i to k . This is the case, for example, when τ_2 is untagged and has no precondition. In general, we aim to merge two transitions by composing their state updates and combining their preconditions and combining their postconditions, while canceling out duplicate facts when it is guaranteed that a fact in the precondition of τ_2 was produced by τ_1 at its postcondition.

There are three main criteria for deciding whether to merge:

First, the tags. If both transitions are tagged, we do not merge them. If τ_1 is tagged, then τ_2 must be unconditional (i.e., have no precondition) to ensure it always follows. If τ_2 is tagged, then τ_1 's postcondition must be local to ensure that it does not cause any side effect on other processes. Note that we do not merge transitions when they are both tagged, as doing so may alter the validity of the correspondence assertions associated with their tags, even though the tags themselves have no computational effect.

Second, the topology around the intermediate node j . We proceed only if either τ_1 is the only incoming edge to j or τ_2 is the only outgoing edge from j . Even when merging is allowed, we remove the original transitions only when it is safe. For example, if j has multiple incoming edges, we can remove τ_1 upon merging, but must retain τ_2 as it may still serve the other incoming transitions.

Third, the preconditions of τ_2 . We merge only when τ_2 has no nonlocal preconditions, in order to preserve its intended behavior of awaiting input from other processes. Even in this case, we avoid merging if both the precondition of τ_2 and the postcondition of τ_1 contain file facts. This is because Rabbit does not forbid the creation of conflicting file facts, so there is no guarantee that a file fact in τ_2 's precondition was produced by τ_1 ; it may have already been in the fact environment.

$$\begin{aligned}
& \langle \Gamma \vdash \text{put}[\vec{A}] \rangle \mathbf{i} = (\{\llbracket \vec{A} \rrbracket_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m)} [\vec{A}]_{\Gamma}^{\mathbf{i}+1}\}, \mathbf{i} + 1) \\
& \langle \Gamma \vdash \text{case } [\vec{A}_1] \Rightarrow c_1 \mid \dots \mid [\vec{A}_n] \Rightarrow c_n \text{ end} \rangle \mathbf{i} = (\bigcup_j \{\llbracket \vec{A}_j \rrbracket_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m)} \llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}_j} \} \cup \\
& \quad \text{let } (G_j, \mathbf{j}) \leftarrow \langle \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \vdash c_j \rangle \mathbf{i}_j \text{ in } G_j \cup \{\llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}} \xrightarrow{(\rho; \Gamma_m)} \llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}+1} \}\}, \mathbf{i} + 1) \\
& \langle \Gamma \vdash \text{repeat } [\vec{A}_1] \Rightarrow c_1 \mid \dots \mid [\vec{A}_n] \Rightarrow c_n \text{ until } [\vec{A}'_1] \Rightarrow c'_1 \mid \dots \mid [\vec{A}'_m] \Rightarrow c'_m \text{ end} \rangle \mathbf{i} = (\{\llbracket \vec{A} \rrbracket_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m), [\text{Loopin}(\kappa, \vec{i})]} \llbracket \vec{A} \rrbracket_{\Gamma}^{\mathbf{i}+1} \} \cup \\
& \quad \bigcup_j \left(\begin{aligned} & \{\llbracket \vec{A}_j \rrbracket_{\Gamma}^{\mathbf{i}+1} \xrightarrow{(\#; \Gamma_m)} \llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}+1} \} \cup \\ & (\text{let } (G_j, \mathbf{j}) \leftarrow \langle \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \vdash c_j \rangle (\mathbf{i} + 1)_j \text{ in } G_j \cup \{\llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}} \xrightarrow{(\#; \Gamma_m), [\text{Loopback}(\kappa, \vec{i})]} \llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}+1} \}) \end{aligned} \right) \cup \\
& \quad \bigcup_j \left(\begin{aligned} & \{\llbracket \vec{A}'_j \rrbracket_{\Gamma}^{\mathbf{i}+1} \xrightarrow{(\#; \Gamma_m)} \llbracket \text{fv}(\vec{A}'_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}+1} \} \cup \\ & (\text{let } (G_j, \mathbf{j}) \leftarrow \langle \text{fv}(\vec{A}'_j), \Gamma_l; \Gamma_m \vdash c'_j \rangle (\mathbf{i} + 1)_{j+n} \text{ in } G_j \cup \{\llbracket \text{fv}(\vec{A}'_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}} \xrightarrow{(\rho; \Gamma_m), [\text{Loopout}(\kappa, \vec{i})]} \llbracket \text{fv}(\vec{A}'_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}+2} \}) \end{aligned} \right), \mathbf{i} + 2) \\
& \langle \Gamma \vdash \text{event}[T] \rangle \mathbf{i} = (\{\llbracket T \rrbracket_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m), [T]} \llbracket T \rrbracket_{\Gamma}^{\mathbf{i}+1} \}, \mathbf{i} + 1) \\
& \langle \Gamma \vdash \text{new } x := S(e_1, \dots, e_n) \text{ in } c \rangle \mathbf{i} = \text{let } (G, \mathbf{j}) \leftarrow \langle x, \Gamma_l; \Gamma_m \vdash c \rangle (\mathbf{i} + 1) \text{ in} \\
& \quad (\{\llbracket \text{Fr}(x) \rrbracket_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m)} [S(\kappa; x, e_1, \dots, e_n)]_{x, \Gamma_l; \Gamma_m}^{\mathbf{i}+1}, \llbracket x, \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}} \xrightarrow{(\rho; \Gamma_m)} \llbracket x, \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}+1} \} \cup G, \mathbf{j} + 1) \\
& \langle \Gamma \vdash \text{let } x_1, \dots, x_n := e.S \text{ in } c \rangle \mathbf{i} = \text{let } (G, \mathbf{j}) \leftarrow \langle x_1, \dots, x_n, \Gamma_l; \Gamma_m \vdash c \rangle (\mathbf{i} + 1) \text{ in} \\
& \quad (\{[S(\kappa; e, x_1, \dots, x_n)]_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m)} [S(\kappa; e, x_1, \dots, x_n)]_{x_1, \dots, x_n, \Gamma_l; \Gamma_m}^{\mathbf{i}+1}, \llbracket x_1, \dots, x_n, \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}} \xrightarrow{(\rho; \Gamma_m)} \llbracket x_1, \dots, x_n, \Gamma_l; \Gamma_m \rrbracket^{\mathbf{j}+1} \} \cup G, \mathbf{j} + 1) \\
& \langle \Gamma \vdash \text{delete } e.S \rangle \mathbf{i} = (\{[S(\kappa; e, x_1, \dots, x_n)]_{\Gamma}^{\mathbf{i}} \xrightarrow{(\#; \Gamma_m)} \llbracket \text{fv}(\vec{A}_j), \Gamma_l; \Gamma_m \rrbracket^{\mathbf{i}+1} \}, \mathbf{i} + 1)
\end{aligned}$$

Fig. 2. Transition graph construction of Rabbit (excerpt)

A special case arises when both transitions contain structure facts of same name, say S , which are introduced precisely to enable cancellations in merging. Suppose τ_2 's precondition includes $S(\kappa; x, x_1, \dots, x_n)$ and τ_1 's postcondition includes $S(\kappa; e, e_1, \dots, e_n)$. Since the second argument (the address) of a structure fact is unique, if $x = e$ can be judged, we conclude that the structure fact in τ_2 is exactly the one produced by τ_1 . Then, we can safely merge the transitions while enforcing $x_i = e_i$, and cancel out the structure facts from both sides. In the case we cannot judge $x = e$, we give up merging the transitions.

Lemma 1. *The compression is well-defined, sound and complete in the sense that it preserves the semantics of systems.*

Proof. We sketch the proof of soundness. First, we prove an invariant that holds for any fact environment that starts from E_0 : for any two facts $S(\kappa; e, \dots)$ and $S(\kappa; e', \dots)$ in E , the values e and e' must be distinct.

Next, we consider a single compression in G_P on edges $\mathbf{i} \rightarrow \mathbf{j}$ and $\mathbf{j} \rightarrow \mathbf{k}$. Let G'_P denote the resulting graph, and Σ' the resulting system. Then it suffices to show that

$$\llbracket \Sigma \rrbracket(\gamma, E) = \llbracket \Sigma' \rrbracket(\gamma, E)$$

for all system states γ such that the local state at P is at $\mathbf{i}$, and for all fact environments E satisfying the invariant. The invariant justifies cancelling out the structure facts. The remaining can be proven easily by a case distinction on the tags and the incoming and outgoing edges of $\mathbf{j}$. $\square$

1) *Example – File System:* Compression is particularly useful for canceling out internal computations with structures. Recall the sequence of commands in Section II-D for opening,

reading, and closing a file. Without any compression, this sequence produces 10 transitions (assuming c is $z := y$, which itself yields one transition). Our compression reduces it to one.

IV. PARAMETERIZATION

A communication channel in Rabbit axiomatizes a connection between processes. To model a situation where a centralized server communicates with unboundedly many arbitrary processes, we require a way to express an unbounded family of channels, and let the server access the entire family, with each channel representing a connection to a specific process.

A **channel family** is denoted by an expression $e\langle \rangle$, where $e\langle e' \rangle$ represents the channel corresponding to index e' in the family. The channel family itself is not an expression unless projected to a concrete index.

A **process family** can be defined using the same language as that for defining a process, but with additional access to a parameter variable, say p , in its expression language. A parameterized client process, for example, can communicate with the server via the channel $\text{udp}\langle p \rangle$. The server, in turn, is connected to the entire channel family $\text{udp}\langle \rangle$ and, using pattern matching in guarded cases,

$$[\text{udp}\langle x \rangle :: \text{Store}(y)] \Rightarrow c$$

can extract the message and the parameter of the client process (to which the logical variable x is bound).

We write $P\langle \rangle$ for a process family, and $P\langle e \rangle$ for a specific process instance whose parameter variable p holds e as its value. Given a finite list of process families $P_1\langle \rangle, \dots, P_n\langle \rangle$, we can form

$$!p. P_1\langle p \rangle \mid \dots \mid P_n\langle p \rangle,$$

to denote a **generator** that unboundedly expands the ambient system, consisting of a finite list of active processes, with $P_1\langle p \rangle, \dots, P_n\langle p \rangle$ for each fresh value of p .

Given a finite list of generators, we assign static names to the process families, for example, $\langle i, j \rangle$ for $P_i\langle \rangle$ when it originates from the j th generator. Using these names, we construct the transition graph for each family of processes, similarly to Figure 2. Most constructions proceed straightforwardly if we treat the parameter variable p as a special local variable, except for the use of the process name κ in structure facts and loop indicators. There, κ is intended to identify the exact process executing the commands, but now all instances of a family share the same κ . Hence, we replace the occurrences of κ in structure facts and loop indicators with $\langle \kappa, p \rangle$. Observe that if a generator always chooses a fresh value for p during instantiation, $\langle \kappa, p \rangle$ serves as a dynamic identifier for the process.

For a finite list of generators as transition graphs, the semantics of a system now includes an additional system transition for introducing new active processes. At any point in a trace, for each generator $!p. P_1\langle p \rangle \mid \dots \mid P_n\langle p \rangle$, and a fresh value v provided by the environment, there exists a system transition that extends the system state with local initial configurations for $P_1\langle v \rangle, \dots, P_n\langle v \rangle$. An internal transition of a newly added process $P_i\langle v \rangle$ then follows the transition graph of $P_i\langle \rangle$.

To simplify the setting, we also treat a normal process without a parameter as a process with a trivial, non-fresh parameter, say $\#$, which simply does not access its parameter. Since such a process cannot be produced multiple times by a generator, the uniqueness property of $\langle \kappa, p \rangle$ across all family names κ and parameters p of active processes still holds.

V. TRANSLATION TO TAMARIN

A prototype implementation of the Rabbit language with a Tamarin backend can be found in the submitted artifact [14] and in the main development repository of Rabbit [15]. The detailed translation mechanism is available there, while this paper focuses on how we achieved tractability of loops.

We translate each transition $\tau : i \rightarrow j$ in the transition graph of a process (family) into a single multiset rewriting rule in Tamarin:

$$[\phi] \multimap [\theta] \rightarrow [\psi]$$

Here, ϕ and ψ are multisets of facts, and θ is a multiset of tags. Similar to the semantics of Rabbit, this rule consumes the linear facts of ϕ from the environment, produces ψ , and tags the action with θ . The key difference between our transitions and Tamarin's multiset rewriting rules is the absence of state indices i . In Rabbit, for a transition to fire, the process must not only satisfy the precondition but also be in the state i . Hence, in the translation, we add **state facts** to both the precondition and postcondition. Each process has its own state facts, with the string representation of the index as an argument, thereby enforcing the structure of our transition graph within the multiset rewriting rules.

Suppose in τ of a graph of a process family named κ , $\Gamma^i = x_1, \dots, x_n; y_1, \dots, y_m$. In the precondition ϕ of the rewriting rule, we add the following state fact:

$$S_\kappa(\langle \bar{i}, p, \xi \rangle, \rho, \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle)$$

The fact is named using the process family's name κ as a suffix. The first argument, $\bar{i}$, ensures that only a process in family κ at state i can apply this transition. The second argument, p , is a parameter variable that captures the parameter value of the process trying to apply this transition. The third argument, ξ , is a variable representing the loop counter. The remaining arguments are variables for the local state.

Suppose $\Gamma^j = x'_1, \dots, x'_k; y'_1, \dots, y'_\ell$. In the postcondition ψ of the rewriting rule, we add a state fact depending on whether the transition τ is looping or not:

- If $\text{Loop}_{\text{back}}$ is the tag of τ , we add:

$$S_\kappa(\langle \bar{j}, p, \xi + 1 \rangle, f_\rho, \langle f_{x'_1}, \dots, f_{x'_k} \rangle, \langle f_{y'_1}, \dots, f_{y'_\ell} \rangle)$$

- Otherwise, we add:

$$S_\kappa(\langle \bar{j}, p, \xi \rangle, f_\rho, \langle f_{x'_1}, \dots, f_{x'_k} \rangle, \langle f_{y'_1}, \dots, f_{y'_\ell} \rangle)$$

The idea is to increment the loop counter each time we loop back, so that a state fact occurs only once for each combination of index, parameter, and loop counter.

We add an auxiliary tag

$$\text{Transition}(\langle \kappa, p \rangle, \bar{i}, \xi)$$

to mark each internal transition of a process with its family name κ , its parameter p , the corresponding program location $\bar{i}$, and the number of loop-backs the process has experienced so far. We later assert that such a transition cannot occur more than once.

A channel fact $e :: F(e_1, \dots, e_n)$ is translated into $F(e, e_1, \dots, e_n)$; a file fact $e_1.e_2$ is translated into $\text{File}_\kappa(p; e_1, e_2)$. The structure fact is translated as-is, and equational facts are handled as how equational theories are treated in Tamarin.

Whenever we translate a channel fact $e :: F(e_1, \dots, e_n)$, either in the pre- or postcondition, we pose an installation fact in the precondition of the rewriting rule to ensure that the process is connected to the channel. These persistent facts are prepared during system initialization.

A. Reuse Lemmas

In Tamarin, a *reuse lemma* is a sub-lemma that can be proved or assumed prior to proving the main assertions. Whether it is proved or assumed, Tamarin's proving engine will use it in main lemmas, and it is sometimes surprisingly useful. Since our generated Tamarin models are specific to Rabbit, we can automatically generate useful reuse lemmas:

- Whenever $\text{Loop}_{\text{Back}}(x, \bar{i})$ or $\text{Loop}_{\text{Out}}(x, \bar{i})$ happens for any x and $\bar{i}$, there is $\text{Loop}_{\text{In}}(x, \bar{i})$ in an earlier moment.
- At a single trace, for any $x, \bar{i}, \xi$, $\text{Transition}(x, \bar{i}, \xi)$ can occur at most once.

Note that the above lemmas cover both parameterized and unparameterized cases, as the variable x may range also over pairs of the form $\langle \kappa, p \rangle$. The first lemma states that whenever a loop is entered back into or exited, there must have been a moment when it was first entered. It is necessary for the proving engine to recognize such control-flow invariants. The second lemma states that each transition happens at most once, up to the loop counter, including the case of nested loops. Though the inner loop may not observe every loop counter value when nested, since the outer loop increments the counter, the premise that each transition occurs at most once for each loop counter value still holds.

Tamarin’s proving engine searches backwards. When a fact is required to occur in a trace that the engine is trying to construct, it considers all rules that could have produced it. This can lead to a combinatorial explosion if loops are involved, or even to nontermination. Our loop-related reuse lemma helps guiding Tamarin by indicating where the loops are. Another scenarios that complicates the backward search is when we have a transition $\tau : i \rightarrow j$ that consumes and reproduces an identical fact. Then, tracing the source of the fact may lead Tamarin to consider arbitrary consecutive executions of the transition τ . Our transition-related reuse lemma, by asserting that a transition τ , up to loop iterations, happens only once, rules out this possibility.

Of course, the idea of introducing loop-related reuse lemmas is not new; it can already be found in [13, Section 10]. Our claim is that, in Rabbit, since loops are primitive constructs, all loops in the system are known even before translation to Tamarin, and the corresponding reuse lemmas can be generated automatically.

An important remark is that these sub-lemmas are guaranteed to hold as long as the translation is correct and can be skipped trusting the translation. They can also be verified by Tamarin as a sanity check for the implementation of the to-Tamarin translator.

B. Main Lemmas

We take the standard way for expressing the reachability and correspondence assertions; see [13, Chapter 6].

VI. EVALUATION

A. The Cam-Image System

As a benchmark example, we consider the cam-image system introduced in [1]. It consists of a centralized server and multiple clients, each connected securely to its Trusted Execution Environment (TEE) component [16], which is responsible for signing the images the client sends to the server. The server then receives the signed images through a UDP channel and verifies the signatures.

To be precise, each iteration consists of:

- 1) A client captures an image from its camera.
- 2) The image is sent to the client’s TEE together with a path in the TEE’s file system for the private key.
- 3) The TEE signs the image using the private key and returns the signed message to the client.

- 4) The client sends the signed image to the server.
- 5) The server verifies the signature using the corresponding public key upon receiving the message.

The connection between the client and its TEE is done via RPC, like a request-acknowledgement mechanism. After some number of iterations, the client sends a message to its TEE to abort the connection.

We consider two assertions: (1) reachability of tags where (i) a client terminates after sending the abort message to the TEE, (ii) a TEE exits the connection with its client, and (iii) the server validates at least one image from the client; and (2) authentication: whenever the server validates a signed image, it must have been sent by a client.

Due to the limited expressivity of the original Rabbit, the system was approximated in both dimensions: the number of iterations and the number of participating entities. It was restricted to only one pair of client and TEE, and the number of iterations was fixed to $n = 1, 2, 3$, where even at $n = 3$, verification was reported no longer feasible.

For the security assertions, the original Rabbit identified an attack scenario breaking authenticity when the UDP channel could be tampered with and the client’s memory could be eavesdropped. We do not explain the scenario in detail here but set it as a benchmark to evaluate whether our extension can still detect the same attack.

With the new command-language of Rabbit, we consider more advanced systems, first without parameterization (the default system) and then with parameterization (the parameterized system). The **default system** works as follows: the client repeats the iteration an unbounded number of times. At the beginning of each iteration, it decides nondeterministically whether to continue. The server is reactive: without a proper exit condition, it runs the image validation loop indefinitely. The TEE is also reactive in that it waits for a message from the client and, depending nondeterministically on the message, either signs and returns the image (if the message is an invoke request) or terminates the loop (if it is an exit message). Recall the examples presented at Section II-E.

The **parameterized system** is the default system but now with an unbounded number of client and TEE pairs. The server listens to the entire family of channels $\text{udp}\langle \rangle$ and a client with parameter p sends its signed image to the server via $\text{udp}\langle p \rangle$.

Unbounded (and nondeterministic) loops, RPC calls, and system calls on the file system in these example systems are expressed as described in Section II-E; e.g., file facts and structure commands are used to model the file system.

B. Evaluation Results

We can verify that our graph compression reduces the number of transitions from 148 to 29 for the default system, and from 124 to 27 for the parameterized system. The parameterized system has fewer transitions because we have slightly simplified some of the internal computations in file systems.

Table I summarizes the time (in minutes, rounded down) required on a Mac Studio (Apple M3 Ultra, 32 cores, 512 GB memory) to prove reachability (t_1), disprove correspondence

TABLE I
EVALUATION RESULTS FOR EXAMPLE SYSTEMS IN RABBIT.

System	Original	Compressed	Compressed&Sub-Lemmas
Default	TO + TO	TO + TO	10 + 25(+TO)
Parameterized	TO + TO	2 + TO	1 + 22(+24)

(t_2), and prove sub-lemmas (t_3 , if applicable), presented in the format $t_1 + t_2(+t_3)$. We evaluated three different configurations of translations: the first is the original, where no optimization technique is applied; the second applies only the graph compression; and the third applies both the graph compression and the assertion of reuse sub-lemmas. In the table, “TO” indicates that the evaluation did not finish within one hour or terminated unexpectedly due to out-of-memory.

The results show that graph compression is necessary for any result to terminate, and it is usually also required to have sub-lemmas. Even though verifying the sub-lemmas timed out for the default system, this is acceptable because the sub-lemmas are guaranteed to be correct and are not a goal of verification. They are advised to be skipped in practice.

Another necessary remark is that both main lemmas require finding a trace, so the evaluation time depends heavily on the trace-finding heuristics. We used the default heuristic provided by Tamarin and do not interpret minor performance differences. Furthermore, the data in the table is not meant for comparison between systems, as the parameterized one has simpler internal computation in TEE.

C. Comparison to SAPIC+ and ProVerif for the file system

As a comparison, we include the results of running the default system using both SAPIC+ (with the default Tamarin backend) and ProVerif. We use the standard encoding of guarded nondeterminism in the π -calculus [17]. For example, repeat $\square \Rightarrow c_1$ until $\square \Rightarrow c_2$ end is encoded as

$$\nu l. \nu e. (1().c_1. \nu x. (\bar{x}() \mid x().\bar{l}() \mid x().\bar{e}())) \mid \bar{l}().e().c_2$$

We observed that both the SAPIC+ model with Tamarin and the ProVerif translation failed to verify a simplified version (without file system) of the default system within one hour.

All Rabbit, SAPIC+, and translated ProVerif files are provided in the artifact [14] to this paper to support reproducibility.

VII. FUTURE WORK

To conclude, we list possible directions for future work.

First is exploring various backend supports. Currently, verification in Rabbit is done only via Tamarin. Similarly to SAPIC+, it would be worthwhile to explore other backends, including ProVerif, based on a different theoretical foundation.

Second is supporting type-based verification by adopting type systems for security assertions from π -calculus-based languages into Rabbit. Introducing data types would also enhance Rabbit’s usability.

Third is the formalization of Rabbit, its semantics, and the correctness proofs of the compression techniques in proof

assistants such as Rocq, Lean, or Isabelle. As Rabbit is intended for use in mission-critical applications, it is desirable for all results, including the Tamarin translator, to be verified.

Fourth is abstraction. Internal computations within a process, carried out using structure facts, can be abstracted by our graph compression, although with some limitations. However, file facts cannot be canceled out, posing a bottleneck in verification. This led us to simplify the TEE’s file system in the parameterized example. We may consider treating file facts similarly to structure facts to support such abstractions.

Regarding abstraction further, we would prefer a way to abstract inter-process computations as well. Ideally, a pair consisting of a Client and its TEE can be abstracted into a single process, provided that their communication channel is secure. This would greatly reduce the verification time and be necessary for scalable modeling of real-world systems.

Fifth is enabling computations. Although Rabbit resembles practical programming languages, it currently only models existing systems and cannot yet be used to implement them. Providing, for instance, a C backend that executes Rabbit programs would allow users to obtain not only verified models but also executable implementations.

Last is combining Rabbit with functional reasoning. Since Rabbit supports general programming constructs, it would be natural to provide mechanisms to reason about the computational behavior of programs, not just their security properties, expanding, e.g., a Hoare-style verification calculus like in [18]

ACKNOWLEDGMENT

The authors would like to thank Yutaka Ishikawa and Taro Sekiyama for helpful discussions. This work was supported by JST, CREST Grant Number JPMJCR21M3.

REFERENCES

- [1] T. Inaba, Y. Ishikawa, A. Igarashi, and T. Sekiyama, “Rabbit: A language to model and verify data flow in networked systems,” in *2024 International Symposium on Networks, Computers and Communications (ISNCC)*, 2024, pp. 1–8.
- [2] J. Jürjens, “UMLsec: Extending UML for secure systems development,” in *UML 2002 — The Unified Modeling Language*, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 412–425.
- [3] J. Jürjens and P. Shabalin, “Automated verification of UMLsec models for security requirements,” in *UML 2004 — The Unified Modeling Language. Modeling Languages and Applications*, T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 365–379.
- [4] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The TAMARIN prover for the symbolic analysis of security protocols,” in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701.
- [5] V. Cheval, C. Jacomme, S. Kremer, and R. Künnemann, “SAPIC+: protocol verifiers of the world, unite!” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3935–3952. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/cheval>
- [6] B. Blanchet, “ProVerif: Cryptographic protocol verifier in the formal model,” <https://proverif.inria.fr/>, 2023, accessed: Apr. 21, 2025.
- [7] E. W. Dijkstra, “Guarded commands, non-determinacy and formal derivation of programs,” *Comm. ACM*, vol. 18, no. 8, pp. 453–457, 1975.
- [8] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, “A meta-notation for protocol analysis,” in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 1999, pp. 55–69.

- [9] V. Cheval, C. Jacomme, S. Kremer, and R. Künnemann, “SAPIC+: A tool for symbolic protocol verification in the applied π -calculus,” <https://sapic-plus.github.io/>, 2022, accessed: Apr. 21, 2025.
- [10] D. Dolev and A. C. Yao, “On the security of public key protocols,” *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.
- [11] M. Abadi and C. Fournet, “Mobile values, new names, and secure communication,” in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’01. New York, NY, USA: Association for Computing Machinery, 2001, p. 104–115. [Online]. Available: <https://doi.org/10.1145/360204.360213>
- [12] G. Lowe, “A hierarchy of authentication specifications,” in *Proceedings 10th Computer Security Foundations Workshop*, 1997, pp. 31–43.
- [13] D. Basin, C. Cremers, J. Dreier, and R. Sasse, “Draft – modeling and analyzing security protocols with Tamarin: A comprehensive guide,” <https://tamarin-prover.com/book/>, 2025, accessed: Apr. 21, 2025.
- [14] S. Park and A. Igarashi, “Rabbit (FMCAD 2025 artifact version),” Aug. 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.16789556>
- [15] Rabbit Development Team, “The development repository of Rabbit,” <https://github.com/zt-iot/rabbit>, ZT-IOT Project, 2025, accessed: Apr. 21, 2025.
- [16] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted execution environment: What it is, and what it is not,” in *2015 IEEE Trustcom/Big-DataSE/ISPA*, vol. 1, 2015, pp. 57–64.
- [17] U. Nestmann and B. C. Pierce, “Decoding choice encodings,” in *CONCUR ’96: Concurrency Theory*, U. Montanari and V. Sassone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 179–194.
- [18] J. Bergstra and C. Middelburg, “Using Hoare Logic in a Process Algebra Setting,” *Fundamenta Informaticae*, vol. 179, no. 4, pp. 321–344, 2021. [Online]. Available: <https://doi.org/10.3233/FI-2021-2026>

Modeling the AWS Authorization Engine

Lee A. Barnett¹, Loris D'Antoni², Amit Goel³, Rami Gökhan Kıcı⁴,

Neha Rungta⁵, Mary Southern⁶, Chungha Sung⁷

Amazon Web Services

Seattle, USA

{lbarnt,lorisd,amgoel,ramikici,rungta,masouth,chunghs}@amazon.com

Abstract—Cloud computing providers employ sophisticated authorization engines to decide when a request to access a resource should be allowed or denied. Several approaches have formalized the behavior of *individual authorization policies*, but authorization engines employ multiple types of policies that can interact in different ways. This paper presents a modular formalization of the Amazon Web Services (AWS) authorization engine and a corresponding analysis tool, called IAM-MULTIPOLICYANALYZER, for verifying properties pertaining to multiple policies of different types. IAM-MULTIPOLICYANALYZER adopts ZELKOVA [18]—i.e., the formalization of individual IAM policies—as a basic building block, and uses a new domain-specific language for modularly describing how the authorization engine composes individual uses of ZELKOVA. As a result, IAM-MULTIPOLICYANALYZER provides a trusted, reusable, human-readable, and performant SMT-backed model of AWS's authorization logic that is now used within multiple AWS applications. We have run conformance testing of our model against the engine implementation and its documentation; the corner cases identified by our testing have led to improvements and modifications to the official AWS documentation.

I. INTRODUCTION

Cloud authorization engines enable secure, on-demand access to digital resources through policies that control who can access what resources and under which conditions. These policies form an essential layer of an organization's security framework, providing clear and enforceable access rules. Specifically, the AWS Identity and Access Management (IAM) policy language [10] governs access by allowing or denying permissions based on rules and conditions on request parameters such as source IP addresses [4] or encryption settings [14]. It provides a rich and expressive syntax that supports multiple policy types, including identity-based policies, resource-based policies, permissions boundaries, and more [6].

To help customers analyze properties of IAM policies, ZELKOVA [18] formalized the meaning of *individual policies* in the AWS IAM language. Using this formalization, IAM policies can be compiled into semantic logical representations that can be analyzed using Satisfiability Modulo Theories (SMT) solvers. For example, ZELKOVA is used to help AWS customers write IAM policies that align with their intent by flagging policies that authorize public access to a resource [21], or by automatically modifying a policy to remove unwanted access in a provably sound way [23].

While ZELKOVA's ability to analyze *individual policies* has found wide adoption and helped many users of AWS,

many security applications require more precise analysis that involve understanding how *multiple policies* interact with each other. This paper formalizes how the AWS authorization engine handles arbitrary combinations of IAM policies, and therefore yields the **first logical model of the semantics of the AWS authorization engine that can be used to automatically analyze combinations of IAM policies**. This model allows one to analyze all authorization decisions for AWS resources, thus allowing new automated analysis within security applications at AWS.

Building a logical model of the semantics of the AWS authorization engine involves striking the right level of granularity:

- 1) *Code is too low level*: The engine is written in a general-purpose language with performance in mind (i.e., it needs to run about one billion authorizations per second [22]), thus directly compiling the code semantics into logical constraints will lead to constraints that are hard to analyze (both in terms of user understanding and SMT solver performance). However, the model should be faithful to the implementation and map cleanly to it.
- 2) *Documentation is too high level*: While the authorization engine is heavily documented in natural language [11], directly translating such a documentation to a logical formalism will yield a logical model that will not map well to the actual authorization code and is therefore hard to test for faithfulness.

To address the considerations above, our work provides a pragmatic approach for building a logical representation that is **analyzable, faithful to the code**, and **well-tested** by carefully threading the needle between the levels of granularity offered by the code and the documentation. By studying the structure of the AWS IAM authorization implementation, we design a *modular SMT-friendly domain-specific language (DSL) for formalizing IAM authorization* that separates modeling into two parts. First, our DSL provides modular constructs that capture the top-level structure of how the actual authorization engine handles multiple policies (e.g., order of evaluation and control flow). Second, the low-level details that pertain to the evaluation of individual policies are encoded as queries to ZELKOVA, which has been used in production for over seven years and heavily tested to be correct.

Thanks to this modular structure, the model of the engine written in the DSL can easily be mapped (line-by-line) to the

actual code. Aside from making the model human-readable, the above mapping allows us to design comprehensive test cases that capture all the combinations of policies that trigger all possible control-flow combinations in the authorization engine. Using these test cases, we can perform conformance testing between the engine implementation and the SMT encoding generated from the model of the engine written in our DSL. To further strengthen our conformance testing, we also consider the third (and customer-facing) way in which the authorization engine is described: the AWS documentation [11]. To automate testing of the documentation, we manually wrote a Scala implementation of the informal semantics described in it.

Our efforts have resulted in three key contributions.

- 1) We built a trusted, human-readable, and performant SMT-backed model of AWS authorization logic that can be used to automatically verify properties of IAM policies across applications (Section III).
- 2) Our modeling strategy has helped generate test cases that were used to improve the official AWS documentation and highlight edge cases customers should consider while authoring policies. (Section IV)

To fully understand the presentation in this paper we recommend a basic familiarity with IAM Policy concepts [6].

II. MODELING AWS AUTHORIZATION

AWS customers can control access to their cloud resources by writing policies in the AWS IAM language [10]. AWS today supports several different kinds of policies, including identity-based and resource-based policies [8], service and resource control policies [16], [15], and permissions boundaries [9]. For example, a resource-based policy (i.e., a policy attached to a resource) can be attached to an Amazon S3 bucket [2] to specify which AWS principals are allowed to read the bucket’s contents. On the other hand, an identity-based policy can be attached to a user, a group of users, or a role, to describe what those identities are allowed to access. For example, an identity-based policy attached to a role can specify resources the role is allowed to access, or which actions the role may perform.

When an attempt to access a resource is made, the AWS authorization engine evaluates all relevant policies, such as the policy attached to the resource and any policies attached to the identity of the principal making the request. The relevant policies, and the final authorization decision (i.e. whether the request is allowed or denied), can depend on the *hierarchical structure of AWS applications*. Resources and identities belong to AWS accounts, and accounts can belong to AWS organizations defining service or resource control policies. Access granted to an AWS account can also be *delegated* to users or roles within that account.

The goal of this paper is to build an analyzable model of the AWS authorization engine. Through an example demonstrating how the authorization engine processes requests involving both identity-based and resource-based policies, we illustrate why

our goal requires a new modeling mechanism that differs from previous work on ZELKOVA.

Example (Cross-Account Authorization): An AWS IAM role [7] named *Ace* is requesting to perform the action *s3:GetObject* on the S3 bucket named *Photo*. *Ace* belongs to the AWS account 111 while the *Photo* bucket belongs to account 222. We consider three cases to demonstrate how the AWS authorization engine evaluates this request in the presence of different policies. Figure 1 shows a diagram of the request and the resulting authorization decisions for each case. We use notation 111/*Ace* to represent a user or role named *Ace* that belongs to the account with identifier 111. This is a short form way of representing Amazon Resource Names (ARNs), which are used within AWS to uniquely identify identities and resources [1].

Case A. The resource-based policy attached to the S3 bucket grants the action *s3:GetObject* to the principal account 111, an alias for the administrator of the account 111, but there is no identity-based policy attached to *Ace*. Because the principal making the request (*Ace*) is in a different AWS account than the resource, permission must also be granted to *Ace* by the policies within the account. Therefore the request is denied.

Case B. An identity-based policy attached to *Ace* grants permission to perform the action *s3:GetObject* on the *Photo* bucket, but there is no resource-based policy. The request is denied, since access is not granted by the bucket’s resource-based policy.

Case C. As in Case B, an identity-based policy attached to *Ace* allows access to the *Photo* bucket. Additionally, the resource-based policy attached to the bucket grants access to the principal account 111. Now the authorization request is allowed: although the bucket’s policy does not name the *Ace* role specifically, the administrator of *Ace*’s account delegates access to *Ace* with the identity-based policy.

The example illustrates how different types of policies can interact in non-trivial ways.

Given that the tool ZELKOVA [18] (described in Section III-A) provides SMT models of the semantics of individual policies, one might wonder if there is an easy way to model the combination of multiple policies using analysis results from ZELKOVA. As we illustrate next, just “combining” the ZELKOVA analyses of different policies cannot soundly capture how different policies interact because permissions can be *delegated* between multiple principals through policies attached to *different entities* (resources, roles, users, etc.).

Consider again the three authorization cases in Figure 1. There are two policies to consider in each case: one for the identity-based policy attached to the *Ace* role in account 111, and another for the resource-based policy attached the *Photo* bucket in account 222. Recall that the request is allowed in Case C, but not in Cases A and B. Let’s analyze what happens if we analyze each policy separately using ZELKOVA, and see if there is a way to combine the results to model the right authorization outcomes.

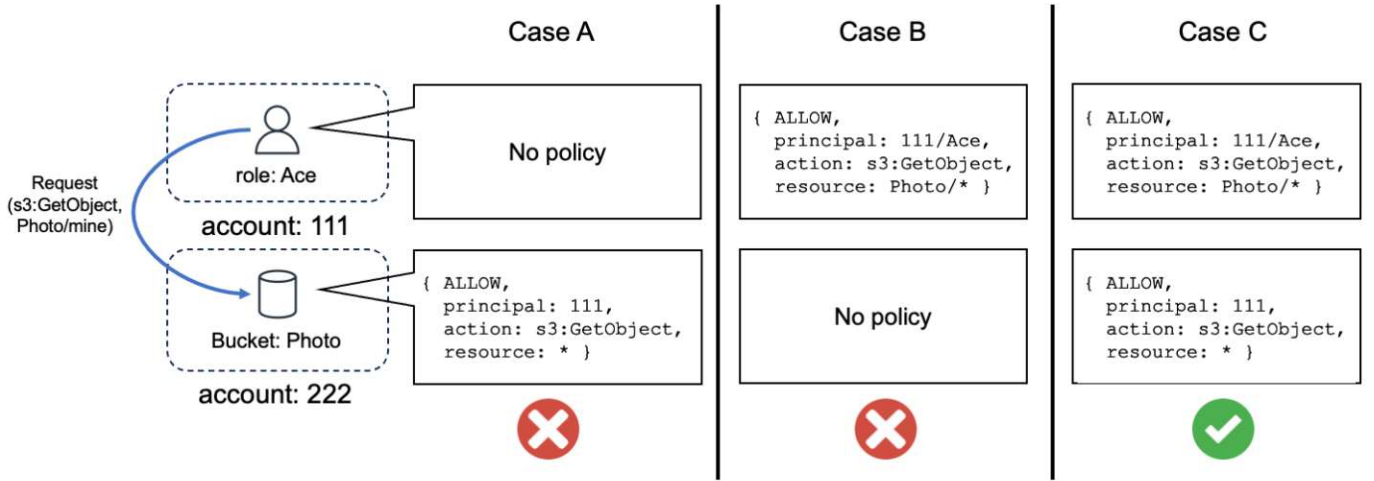


Fig. 1. Example of AWS authorization for two types of policies.

For Case A, ZELKOVA captures that the principal in the request does not match the permissions granted by the resource-based policy—i.e., the resource-based policy only grants access to principal 111, with no mention of the case in which `Ace` role makes the request. Similarly, in Case B, ZELKOVA captures the fact that the identity-based policy allows the request but the resource-based policy denies—i.e., an empty policy does not allow any access.

From Case A and Case B, one might guess that the actual authorization result is `Allow` if and only if ZELKOVA concludes that both the identity-based and resource-based policies result in an `Allow` decision. However, such an approach is *not sound* (i.e., the model would not match the actual implementation): in Case C the request should be allowed, analyzing the resource-based policy in isolation using ZELKOVA would yield the same result as for Case A—i.e., that the request should not be allowed.

A similar issue can arise when a resource-based policy allows access to a specific role in the same account as the resource, but an identity-based policy in that role’s account explicitly denies access. ZELKOVA’s per-policy analysis might suggest allowing access (seeing the resource-based allow), while the actual AWS authorization engine would correctly deny it due to the explicit deny in the identity-based policy.

These examples demonstrate how one cannot model the AWS authorization engine by separately analyzing single policies with ZELKOVA and taking Boolean combinations of the analysis outcomes. The key missing element is the ability to model how *permissions flow through delegation relationships between principals*. A sound analysis needs to represent the hierarchical structure of AWS principals and resources, capturing how permissions are delegated from resources to accounts and from accounts to roles or users. The tool presented in this paper, IAM-MULTIPOLICYANALYZER, introduces a DSL that still uses ZELKOVA as a building block, but explicitly

models permissions delegation and uses it to soundly capture all the features of the AWS authorization engine, including cross-account access requirements and the precedence of different policy types. In IAM-MULTIPOLICYANALYZER, the authorization decision for Case C is modeled using the DSL program illustrated in Figure 3. While we discuss the program in Figure 3 in detail in Section III, the key idea behind IAM-MULTIPOLICYANALYZER’s model in this DSL is that delegation is treated explicitly by issuing separate calls to ZELKOVA using different principals (as illustrated in the third let-statement).

III. A MODULAR FORMAL MODEL FOR AWS AUTHORIZATION

Because our goal is to build a model of **any possible combination of all types of policies** with allow and deny statements, modeling the authorization engine requires a new modular mechanism to combine the decisions performed by individual policies. In this section, we present IAM-MULTIPOLICYANALYZER, a modular approach for translating sets of AWS policies into human-readable SMT terms that exactly capture the policies’ semantics according to the AWS authorization engine implementation.

IAM-MULTIPOLICYANALYZER relies on ZELKOVA, a model for single policies that has withstood the test of time, to model individual policies (Section III-A) and a high level SMT friendly domain-specific language (DSL) that models the control flow of how the authorization engine dispatches authorization to individual policies and combines their results (Section III-C).

A. ZELKOVA: From Individual Policies to SMT

The key component that enables modularity is ZELKOVA [18], an existing service that compiles the semantics of individual IAM policies into SMT. The core of ZELKOVA’s compilation is the deconstruction of policies into

$$p = \text{"111/Ace"} \wedge a = \text{"s3:GetObject"} \wedge r \text{ matches } \text{"Photo/*"} \quad (1)$$

$$p = \text{"111"} \wedge a = \text{"s3:GetObject"} \quad (2)$$

Fig. 2. ZELKOVA SMT Encoding of the Identity-based Policy (Equation (1)) and the Resource-based Policy (Equation (2)) from Case C of Figure 1.

a set of statements, where each statement has effect (ALLOW or DENY), principal, action, and resource components. Conditions can optionally be used to further restrict which requests a policy allows, but are not described here for space considerations (details of conditions in ZELKOVA are described by Backes et al. [18]).

The ZELKOVA SMT encoding of each policy statement s is a formula $\varphi_s(p, a, r)$ over three variables: principal (p), action (a), and resource (r). Satisfying assignments to the formula $\varphi_s(p, a, r)$ correspond to requests for which the policy grants access (i.e., what principals can perform what actions on what resources). Figure 2 shows the ZELKOVA SMT encodings of the two policies shown in Case C of Figure 1. In Equation (1), the principal and action must match the specific concrete values given in the policy, while the resource must match the expression “Photo/*” containing a wildcard $*$ (indicating that it may be any string that begins with “Photo/”).

To meet the performance required by its applications within AWS, ZELKOVA employs many powerful optimizations, and taking advantage of such optimizations is one of the key design choices behind the modular approach of IAM-MULTIPOLICYANALYZER. Concretely, ZELKOVA supports many optimizations for policies involving string and set comparison, which we want to reuse in our design. For example, in Equation (2), the resource component is omitted as a ZELKOVA optimization detects that any string would match the regular expression containing only the wildcard.

When multiple statements appear in *the same policy*, the SMT formula produced by ZELKOVA is a conjunction of: (1) a disjunction of those statements with effect ALLOW, and (2) a conjunction of the negation of the statements with effect DENY—i.e., a request is allowed if it is explicitly allowed by some ALLOW statement and is not denied by any DENY statement. As discussed in Section II, this Boolean-combination approach is only sound (i.e., it accurately captures whether a request should be allowed) when considering a single policy, but fails when two statements belong to policies of different types.

B. AUTHENGINE: AWS Authorization Engine

In AWS, the authorization engine (which we call AUTHENGINE) handles the policy evaluation logic. At a high level, AUTHENGINE takes as input a set of policies $\{P_1, \dots, P_n\}$ and a request r including details about the principal, action, resource, and the context about the environment/resource (e.g., IP address) [13], and decides whether the request should be allowed.

Algorithm 1 AUTHENGINE algorithm snippet for Section II

Require: req is the authorization request for role Ace ; req_A is the authorization request for account 111; $evaluate$ evaluates a policy given a request and returns *Allow*, *ImpDeny* or *Deny*; P_I is the identity policy; P_R is the resource-based policy

Ensure: Returns *true* iff cross-account access request is allowed by policies

if $evaluate(P_R, req_A) = Deny \vee evaluate(P_R, req) = Deny \vee evaluate(P_I, req) = Deny$ **then**
 return false ▷ Deny trumps all

end if

$delegated \leftarrow evaluate(P_R, req_A) = Allow$ ▷ Trusting account delegates to the trusted account

$trusting \leftarrow evaluate(P_R, req) = Allow$

$trusted \leftarrow evaluate(P_I, req) = Allow$

return $trusted \wedge (trusting \vee delegated)$

The key mechanism employed by the AUTHENGINE (and the one that we model explicitly in our DSL) is the *delegation* of permissions across principals. This mechanism allows permissions to flow automatically through chains of principals (e.g., from organization to account to role) without requiring explicit definition of each transfer. This mechanism is enabled by the interaction between the so-called trusted and trusting principals, where if a trusting principal A , e.g. an account, can access a resource and delegates permission to principal B , e.g. a role within that account, then B becomes a trusted principal who can further delegate those permissions down the chain [5].

A common and practical use case for delegation is inter-account access, where the authors of the resource-based and identity-based policies are different actors. Using permission delegation, a resource policy can allow access to a trusted account without the resource owner needing to know the details of the identities within that account. The trusted account is then responsible for further delegating appropriate permissions to the users or roles that should have access. A key point is that a principal is only able to delegate the access they themselves have. For example, if a role A cannot access a resource R , that same role cannot delegate access to resource R to any principal.

While the internals of AUTHENGINE are closed-source and cannot be described in detail, the reader can refer to AWS documentation for an overview of AUTHENGINE [12]. Additionally, Algorithm 1 in contains a simplified snippet that corresponds to how AUTHENGINE would operate for the case described in Figure 1.

C. IAM-MULTIPOLICYANALYZER: From Sets of Policies to SMT via ZELKOVA

In this section, we describe how IAM-MULTIPOLICYANALYZER models multiple policies and delegation of permissions. IAM-MULTIPOLICYANALYZER follows two key principles:

- *Modularity*: IAM-MULTIPOLICYANALYZER should use ZELKOVA (a well-tested and optimized trusted component) as a key primitive whenever it models an individual policy.
- *Faithfulness*: IAM-MULTIPOLICYANALYZER’s model should follow the original implementation of AUTHENGINE as closely as possible—i.e., we should be able to map each line of the model to a corresponding line in AUTHENGINE.

IAM-MULTIPOLICYANALYZER uses a DSL (Figure 4) to model how permissions are delegated by the AUTHENGINE. Concretely, given a set of policies $\mathcal{P}$, IAM-MULTIPOLICYANALYZER builds a program X in the DSL (which can then be translated to an SMT formula), such that a request r is a valid model of X iff AUTHENGINE allows the request r for policies $\mathcal{P}$. In the rest of the section, we describe the components of the DSL.

1) *Main Program*: A program in the IAM-MULTIPOLICYANALYZER DSL is a predicate that describes an authorization decision. Complex predicates can be built using let-statements, Boolean operations, and most importantly, the three specialized predicates used for handling delegation listed under `IRPredicate` in Figure 4.

For example, the snippet of DSL program in Figure 3 models how AUTHENGINE handles the policies in Case C of Section II. The first let expression models the access through some identity policy which is attached to the principal, in this case `111/Ace`. The second let expression models requests allowed by the resource-based policy directly. The final let expression models that a resource may delegate permissions through the account when the acting principal has an appropriate principal type (`Role` or `User`). To keep the example manageable we do not include the clause for principal types with more complex delegation chains, such as `AssumedRole`, as our running example requires only the `Role` type. However, IAM-MULTIPOLICYANALYZER does support these multi-jump delegation chains as well. This example demonstrates how this DSL creates a human-readable program that expresses the model of access control in AUTHENGINE and is closely aligned with the authorization algorithm (Algorithm 1).

We now describe each specialized predicate and how their semantics is translated to SMT terms to enable automated reasoning on policies.

2) *Evaluation under Context*: The main primitive in the DSL is the predicate `EvalCtx(IRCtx, AuthPolicy, Outcome)`, which is used to dispatch the modeling of individual policies to ZELKOVA. This predicate has the following intended behavior (i.e., set of satisfying assignments): under context `IRCtx`, the *single policy* `AuthPolicy` evaluates to the outcome `Outcome`.

As we have seen in Section III-A, single-policy evaluation results in a formula $\varphi(p, a, r)$. The context `IRCtx` represents the variable valuations relevant to this single policy evaluation within the larger evaluation—e.g., who is the principal p , what

```

Let AllowI =
  PrHasArn({111/Ace, a, r}, 111/Ace) ∧
  EvalCtx({111/Ace, a, r}, PI, ALLOW)
in
Let AllowR =
  EvalCtx({111/Ace, a, r}, PR, ALLOW)
Let AllowDelegateR =
  (ActPrHasType({111/Ace, a, r}, Role) ∨
   ActPrHasType({111/Ace, a, r}, User)) ∧
  EvalCtx({111, a, r}, PR, ALLOW)
in
  AllowI ∧ (AllowR ∨ AllowDelegateR)

```

Fig. 3. DSL program modeling the authorization of Case C from Figure 1 in IAM-MULTIPOLICYANALYZER.

resource r are they accessing, etc. For simplicity we will restrict the presentation of `IRCtx` to just the principal, action, and resource; eliding condition keys. Intuitively, an `IRCtx` is a predicate acting as a symbolic representation of a set of actual concrete requests.

To model delegation, the `IRCtx` will have different principal “assignments” when performing different evaluations. For example, the two instances of `EvalCtx` in the second and third let statements in Figure 3 use different contexts `{111, a, r}` and `{111/Ace, a, r}` to model the two possible delegation paths for the role `111/Ace`. The resulting authorization decision is the disjunction of the two predicates.

To summarize, a satisfying assignment for a predicate `EvalCtx(IRCtx, P, Allow)` is any evaluation context that is consistent with the predicate `IRCtx` and that results in an allow-decision when evaluated by AUTHENGINE on the single policy P . While in our example, the principal value is set to different constants corresponding to the possible delegated principal values in each `IRCtx`, it is also possible to assign delegated principal values symbolically through predicates over p .

3) *Predicates for Constraining Principals*: While the snippet illustrated in Figure 3 only contains the case in which the acting principal has type `Role` or `User`, the full model of the example in Figure 1 will also contain cases for other types of delegation, with the corresponding evaluation behavior.

In AUTHENGINE, the delegation mechanism depends on the type of a principal (e.g., `Root`, `User`, `Role`) and the structure of its ARN (a format used by AWS to locate resources). It is not important to understand the specifics of ARNs for this work and so specific discussion of ARNs is left to the existing AWS documentation [1]. For the sake of this explanation, it is just relevant to know that AWS imposes constraints of what structures of ARNs one can use for different types of principal types.

To correctly model delegation, the DSL uses two pred-

```

IRExpr ::= Let id = IRExpr in IRExpr |
  IRPredicate |
  TRUE | FALSE |  $\neg$ IRExpr |
  IRExpr  $\wedge$  IRExpr | IRExpr  $\vee$  IRExpr

IRPredicate ::=
  EvalCtx(IRCtx, AuthPolicy, Outcome) |
  PrHasArn(IRCtx, Arn) |
  ActPrHasType(IRCtx, PrncplType)

PrncplType ::=
  Root | Role | User |
  AssumedRole | FederatedUser |
  FederatedPrincipal

```

Fig. 4. Simplified Syntax of IAM-MULTIPOLICYANALYZER DSL.

icates, `PrHasArn` and `ActPrHasType`, that capture the requirements of different uses of principal in AUTHENGINE. The predicate `PrHasArn` (`IRCtx`, `Arn`) holds when the acting principal in `IRCtx` has the ARN `Arn`, while `ActPrHasType` (`IRCtx`, `PrncplType`) holds when the acting principal in `IRCtx` has type `PrncplType`.

In our example, the principal is a constant and therefore one can trivially infer the account and roles, but in general one may require string predicates to do so. In fact, while Figure 3 only illustrates the snippet of DSL program corresponding to the case in which the principal has type `Role` or `User`, IAM-MULTIPOLICYANALYZER will generate similar snippets for other types of principals. Because for such snippets the predicate `ActPrHasType` will be false, these snippets will never affect the authorization decision in this case.

4) Translation to SMT: For an IAM-MULTIPOLICYANALYZER model (i.e., a program in the DSL) to be analyzable, we must generate an SMT term corresponding to it. We cannot share the exact details in this work, but implementing this translation requires maintaining the shared variable state across different ZELKOVA invocations issued by `EvalCtx` to ensure consistency across the dependent evaluations. Intuitively, if `irCtx` is the `IRCtx` and φ is the formula encoding the ZELKOVA invocation, then the formula $\varphi(irCtx)$ encodes the IR expression `EvalCtx(irCtx, P, ALLOW)`. The final SMT formula is otherwise generated by composing ZELKOVA SMT terms according to the structure of the DSL term. The check of whether the request from Figure 1 matches the DSL program in Figure 3 is captured by the following SMT

formula.

$$\left(\begin{array}{c} \text{"111/Ace"} = \text{"111/Ace"} \wedge \\ a = \text{"s3:GetObject"} \wedge \\ r \text{ matches } \text{"Photo/*"} \end{array} \right) \wedge \\
 \left(\left(\begin{array}{c} \text{"111/Ace"} = \text{"111"} \wedge \\ a = \text{"s3:GetObject"} \wedge \\ r \text{ matches } \text{"*"} \end{array} \right) \vee \left(\begin{array}{c} \text{"111"} = \text{"111"} \wedge \\ a = \text{"s3:GetObject"} \wedge \\ r \text{ matches } \text{"*"} \end{array} \right) \right)$$

The generated SMT formula has the desired behavior described in Section II: any request with principal 111/Ace to perform action `s3:GetObject` on the Photo bucket corresponds to an instantiation which makes this formula true. The vacuous string equalities are due to the fact that we are instantiating the principal with a specific constant.

Using an intermediate DSL to build SMT models enables multiple important optimizations in IAM-MULTIPOLICYANALYZER. The system applies constant folding to simplify expressions, caches ZELKOVA models of individual policies for reuse, and filters irrelevant statements from resource-based and resource-control policies based on the requesting principal. The DSL approach also produces human-readable models, as shown in Figure 3, making it straightforward to confirm that they capture the AUTHENGINE logic.

IV. EVALUATION

In this section we evaluate IAM-MULTIPOLICYANALYZER by assessing its correctness with respect to AUTHENGINE, as well its performance and usability within applications in Amazon.

In Section IV-B we evaluate the faithfulness of IAM-MULTIPOLICYANALYZER to the implementation of AUTHENGINE, additionally comparing against the publicly available documentation about AWS authorization, using a three-way conformance testing approach. In Section IV-C we examine applications within Amazon using IAM-MULTIPOLICYANALYZER and in Section IV-D we assess its runtime performance responding to service requests.

A. Implementation

IAM-MULTIPOLICYANALYZER intentionally relies upon the existing ZELKOVA model of single-policy semantics, which has been available through an Amazon internal web service used by tens of applications for several years. We implemented IAM-MULTIPOLICYANALYZER as a significant addition on top of ZELKOVA, extending this web service with novel operations that use IAM-MULTIPOLICYANALYZER’s model to check properties about the effective permissions of sets of policies; for example, whether any kind of access is possible from a specific principal to an S3 Bucket given all associated policies. Like ZELKOVA, IAM-MULTIPOLICYANALYZER runs on AWS Lambda and relies upon a portfolio of solvers comprising Z3 [24], CVC4 [20], CVC5 [19], and NFA2SAT [29]. IAM-MULTIPOLICYANALYZER was designed to reuse existing ZELKOVA code as much as possible,

so that it benefits from the correctness and efficiency already built into ZELKOVA, as well as any future improvements made to ZELKOVA.

B. Conformance of the Model to the Code and Documentation

We performed 3-way conformance testing of the following implementations:

- *IAM-MULTIPOLICYANALYZER's model*: the model G contributed by this paper.
- *AUTHENGINE implementation*: the Java implementation I of the AWS authorization engine modeled in this paper.
- *Informal IAM documentation*: a Scala program D encoding the informal specification of the AWS authorization logic presented in the IAM documentation [12]. Like I , the program D takes as input AWS policies along with an authorization request, and returns a yes/no answer representing an authorization decision. Unlike I , it does not include any optimizations, intended only to correspond cleanly to how AWS authorization is explained in the documentation. Note that D is quite distinct from G : it does not produce an SMT formula or other formal model of authorization, but is simply a less efficient version of I .

To guarantee complete and exhaustive evaluation of conformance, we need test inputs that exercise all control-flow paths of the AUTHENGINE implementation. Each test input should include a set of policies P , also specifying the entities to which they are attached, as well as an authorization request r . We say that the three systems G , I , and D are consistent on a given test (P, r) if they all agree on whether the request r should be allowed by the set of policies P . However, generating such inputs is a hard task because (1) randomly generating pairs of the form (P, r) results in many meaningless policies and requests, whereas (2) automated symbolic (or concolic) execution is difficult due to the complexity of the involved datatypes; i.e., complex JSON objects with string variables.

To sidestep this problem, we automatically generated tests that consider only the dimensions of policies that could affect the overall control-flow of how an authorization request r is evaluated. Specifically, we wanted to account for whether each policy in the set P would individually allow or deny the request r , but did not want to account for all the many different ways in which one policy could be written to allow or deny r . We think this design choice is justified by the fact that IAM-MULTIPOLICYANALYZER relies on ZELKOVA's trusted model of single-policy semantics (i.e., single-policy semantics have already been heavily tested). Hence, we were able to focus testing on the high-level policy combining logic within AUTHENGINE without worrying about the details of intra-policy evaluation.

Concretely, our generated test cases comprise all potential combinations of policy types as well as combinations of whether policies allow, implicitly deny, or explicitly deny potential principals. The final set of test cases contained approximately 1,500 (P, r) pairs.

Findings. IAM-MULTIPOLICYANALYZER's model G and AUTHENGINE's implementation I agreed on all the test cases, thus providing us with high assurance that IAM-MULTIPOLICYANALYZER's model is sound. Additionally, our tests highlighted two edge cases that the Scala program D did not capture, and have since resulted in changes to IAM documentation [3]. These changes improve the description of authorization presented to AWS customers. For example, one of these edge cases showed that the documentation did not fully capture how the "NotPrincipal" policy element in resource-based policies works in combination with permission boundary policies. Today, the documentation contains a clear warning highlighting that customers should avoid including a NotPrincipal policy element with a Deny effect for IAM users or roles that have a permissions boundary policy attached.

C. Applications of IAM-MULTIPOLICYANALYZER

We discuss three applications running within Amazon that depend on IAM-MULTIPOLICYANALYZER, chosen to demonstrate the range of use-cases for which this precise, multi-policy model of authorization is beneficial.

Security When necessary, Amazon may store customer data, which can in some cases include highly-sensitive information such as payment and health records. Amazon's top priority in handling this data is keeping it secure, ensuring it is available only to the small number of applications where it is absolutely needed. One of the ways in which access to sensitive data is safeguarded is through a security tool that uses IAM-MULTIPOLICYANALYZER to audit permissions granted to these critical resources by AWS policies. The tool sends regular requests to the IAM-MULTIPOLICYANALYZER web service to determine all principals that are granted permission to access any of this sensitive data, considering the identity-based policies of those principals in combination with the critical data's resource-based policies and any organization-level policies. IAM-MULTIPOLICYANALYZER enables the security tool to produce a complete picture of all possible ways access could be granted to these sensitive resources by AWS policies, helping detect possible security risks.

Availability Service-control policies (SCPs) are guardrails that define the maximum permissions which can be granted to members at different levels of an AWS Organization. Modifications to SCPs can have large-scale effects, possibly affecting access for all principals within an organizational unit. IAM-MULTIPOLICYANALYZER is used by a tool within Amazon that checks whether proposed changes to their organization's SCPs are safe before applying them. The tool sends requests to the IAM-MULTIPOLICYANALYZER web service that include the proposed new SCPs along with identity-policies for principals to ensure that (1) the new SCPs do not permit new accesses that should never be allowed, and (2) the new SCPs do not cause an availability risk by preventing needed permissions that are necessary for their applications' intended functions.

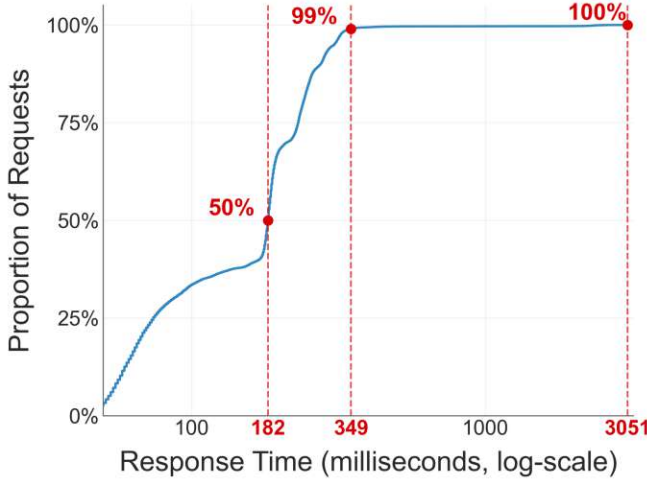


Fig. 5. Performance of IAM-MULTIPOLICYANALYZER on one million policy analysis requests.

Automation Some services within AWS help customers onboard to their cloud offerings by automatically performing actions on behalf of the customer, such as the deployment of new resources within their account. This automation can be done by assuming a role with permissions to perform these actions in the customer’s account, but in some cases there can be additional policies such as SCPs or resource-based policies that prevent the automation from succeeding. One service that relies heavily on this automation uses IAM-MULTIPOLICYANALYZER to determine whether their automation will succeed before attempting it, calling the IAM-MULTIPOLICYANALYZER web service with all relevant policies to check that each needed action is allowed. This check enables them to work with customers ahead of time to identify any policies that could cause problems for the onboarding, reducing friction around the adoption of their service.

D. Performance

General metrics The total number of invocations of IAM-MULTIPOLICYANALYZER ranges from hundreds of thousands to over a million in a single day. Figure 5 shows the performance of the analysis on a randomly-chosen set of one million requests to IAM-MULTIPOLICYANALYZER from multiple services. The y-axis represents the percentage of requests solved within the time on the x-axis. The graph shows that 99% of requests are solved within 349 milliseconds, with a median response time of 182 milliseconds.

Comparison to prior approaches While ZELKOVA can answer questions about what could be allowed by a single policy, IAM-MULTIPOLICYANALYZER is used to analyze the effective permissions of multiple policies with respect to AUTHENGINE. The only policy-analysis problems both tools can solve are ones involving single policies, but in this setting,

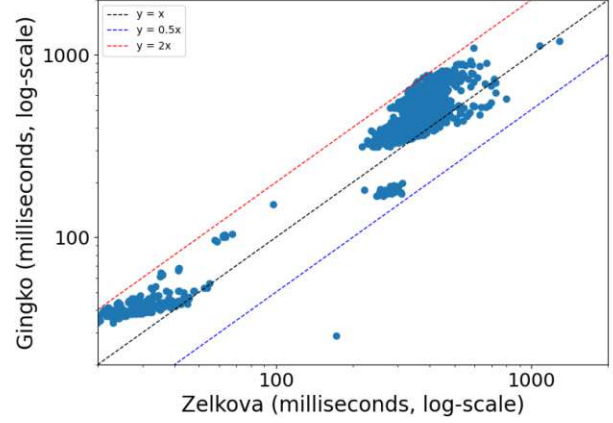


Fig. 6. Performance of IAM-MULTIPOLICYANALYZER compared to ZELKOVA on 3819 security tool requests involving identity-based and service control policies.

the two tools are identical as IAM-MULTIPOLICYANALYZER simply encodes a single policy using ZELKOVA.

To perform a meaningful comparison between the two tools, we retrofit a set of benchmarks that were created by the security tool described in Section IV-C, which was in fact using ZELKOVA before the development of IAM-MULTIPOLICYANALYZER. Though ZELKOVA is intended to only analyze a single policy at a time, this application relied on custom compositions of calls to ZELKOVA that combined multiple policies into one for analysis (i.e., they manually implemented a special case of the IAM-MULTIPOLICYANALYZER’s semantics). The tool’s ad hoc policy-combining logic applied several heuristics to improve the performance of ZELKOVA on their requests.

When using IAM-MULTIPOLICYANALYZER to perform the same analysis, we observed no differences in the outcomes of the analysis (i.e., the two tools agreed in all cases) and observed similar performance. Figure 6 shows a scatter plot of the performance of IAM-MULTIPOLICYANALYZER on 3,819 requests from the security tool, compared to the performance of the existing ad-hoc implementation using ZELKOVA. On average, the response time of IAM-MULTIPOLICYANALYZER on this set of requests was 99.94 milliseconds (or 1.31 times) longer than the response time of ZELKOVA, with 99% of requests taking at most 331.90 milliseconds (or 1.87 times) longer than the response time of ZELKOVA.

While the ad-hoc ZELKOVA encoding could take advantage of the problem structure to achieve better performance, the security tool still decided to switch to IAM-MULTIPOLICYANALYZER because (1) conformance testing provides assurance that IAM-MULTIPOLICYANALYZER faithfully captures the AWS authorization logic, and (2) the performance was largely comparable.

Furthermore, because IAM-MULTIPOLICYANALYZER can model more policy types than just identity-based policies and

service control policies, this tool was able to extend its usage to include additional policy types, e.g., resource-based policies, thus enabling more precise and generalizable analyses.

V. RELATED WORK

ZELKOVA [18] has been used in a variety of applications. For example, the *stratified predicate abstraction* algorithm was built on top of ZELKOVA to summarize all permissions granted by an IAM policy [17], while Bouchet et al. [21] introduced an approach built on ZELKOVA to detect IAM policies that grant overly broad or *public* access. These extensions focus on identifying which permissions are allowed or denied by single IAM policies; in a similar vein, D’Antoni et al. [23] describe a technique for automatically modifying IAM policies to remove undesired permissions.

While ZELKOVA models the permissions granted by individual access control policies in isolation, IAM-MULTIPOLICYANALYZER focuses on the combined effect of multiple policies, which may be attached to distinct, distributed cloud resources, on authorization decisions. The applications that use ZELKOVA can now be lifted to supporting multiple policies by adopting IAM-MULTIPOLICYANALYZER.

Other approaches have also looked at modeling permissions granted by access control policies expressed in different languages. Fisler et al. [26] use transitions between evaluation states to determine access control in policies. Other applications also use an SMT encoding to model authorization, such as the work on verification of NGAC policies by Dubrovenski et al. [25]. These applications target different languages and authorization mechanisms and cannot be applied to our domain. SecGuru [28] also uses SMT; specifically, the SMT theory of bit vectors, to compare network connectivity policies. Hughes and Bultan [27] transform XACML policies into Boolean satisfiability problems and use a SAT solver to check partial orders between policies using a bounded analysis. This bounded analysis is however unsound, while both ZELKOVA and IAM-MULTIPOLICYANALYZER are based on a sound SMT encoding that ensures all allowed accesses are identified.

VI. CONCLUSION

We proposed a modular approach for analyzing how AWS IAM authorization handles sets of policies and implemented it in IAM-MULTIPOLICYANALYZER. IAM-MULTIPOLICYANALYZER has launched as an internal service in AWS, serving about two million requests per week showing the scale at which the approach operates in practice. Customers use IAM-MULTIPOLICYANALYZER in many tasks and applications, from checking that the service control policies used within their organizations are not overly permissive, to detecting ahead of deployment when overly-restrictive service control policies can cause access-denied issues. As next steps, we are building solutions that use IAM-MULTIPOLICYANALYZER for AWS customers—e.g., allowing customers to audit who has access to what within their accounts and organizations.

REFERENCES

- [1] Amazon resource names. Accessed: January 2025. URL: <https://docs.aws.amazon.com/IAM/latest/UserGuide/reference-arns.html>.
- [2] Amazon simple storage service documentation. <https://docs.aws.amazon.com/s3/>. Accessed: January 2025.
- [3] AWS JSON policy elements: NotPrincipal. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_notprincipal.html.
- [4] Denies access to AWS based on the source IP. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_aws_deny-ip.html.
- [5] Examples of policies for delegating access. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create_policy-examples.html.
- [6] IAM JSON policy reference. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies.html.
- [7] IAM roles. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html.
- [8] Identity-based policies and resource-based policies. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_identity-vs-resource.html.
- [9] Permissions boundaries for IAM entities. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_boundaries.html.
- [10] Policies and permissions in AWS identity and access management. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.
- [11] Policy evaluation logic. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html.
- [12] Policy evaluation logic. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html.
- [13] Processing the request context. Accessed: January 2025. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic_policy-eval-request-context.html.
- [14] Protecting data with server-side encryption. Accessed: January 2025. URL: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/serv-side-encryption.html>.
- [15] Resource control policies (RCPs). Accessed: January 2025. URL: https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_rcps.html.
- [16] Service control policies (SCPs). Accessed: January 2025. URL: https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html.
- [17] J. Backes, U. Berruco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan. Stratified abstraction of access control policies. In *Computer Aided Verification—CAV*, pages 165–176, 2020. doi:10.1007/978-3-030-53288-8_9.
- [18] J. Backes, P. Bolognani, 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 *Formal Methods in Computer Aided Design—FMCAD*, pages 1–9, 2018. doi:10.23919/FMCAD.2018.8602994.
- [19] 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—TACAS*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.
- [20] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd Intl Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.
- [21] 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*

- [22] A. Chakarov, J. Geldenhuys, M. Heck, M. Hicks, S. Huang, G. A. Jaloyan, A. Joshi, R. Leino, M. Mayer, S. McLaughlin, A. Mritunjai, C. P. Claudel, S. Porncharoenwase, F. Rabe, M. Rapoport, G. Reger, C. Roux, N. Rungta, R. Salkeld, M. Schlaipfer, D. Schoepe, J. Schwartzentruber, S. Tasiran, A. Tomb, E. Torlak, J. Tristan, L. Wagner, M. Whalen, R. Willems, J. Xiang, T. J. Byun, J. Cohen, R. Wang, J. Jang, J. Rath, H. T. Syeda, D. Wagner, and Y. Yuan. Formally verified cloud-scale authorization. 2025. URL: <https://www.amazon.science/publications/formally-verified-cloud-scale-authorization>.
- [23] L. D’Antoni, S. Ding, A. Goel, M. Ramesh, N. Rungta, and C. Sung. Automatically reducing privilege for access control policies. 8, Oct. 2024. doi:10.1145/3689738.
- [24] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [25] V. Dubrovski, E. Chen, and D. Xu. SMT-based verification of NGAC policies. In *IEEE Computers, Software, and Applications Conference—COMPSAC*, pages 860–869, 2023. doi:10.1109/COMPSAC57700.2023.00115.
- [26] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings 27th Intl Conference on Software Engineering—ICSE.*, pages 196–205, 2005. doi:10.1109/ICSE.2005.1553562.
- [27] G. Hughes and T. Bultan. Automated verification of access control policies using a SAT solver. *Intl journal on software tools for technology transfer*, 10(6):503–520, 2008.
- [28] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman. Automated analysis and debugging of network connectivity policies. *Microsoft Research*, 1:1–11, 2014.
- [29] 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—CAV*, pages 187–208, 2023.

Automated Translation Validation of a Compiler for Statically Scheduled Accelerators

Jackson Melchert	Caleb Terrill	Aron Ricardo Perez-Lopez	Clark Barrett	Priyanka Raina
Stanford University	Stanford University	Stanford University	Stanford University	Stanford University
Stanford, United States	Stanford, United States	Stanford, United States	Stanford, United States	Stanford, United States
melchert@stanford.edu	cterrill@stanford.edu	arpl@cs.stanford.edu	barrettc@stanford.edu	praina@stanford.edu

Abstract—Compilers for programmable hardware accelerators are complex and involve progressively lowering an application down to the hardware. Bugs can be introduced at many different stages, but simulation does not provide full bug coverage and has poor bug localization. We propose a methodology for automated formal translation validation of compilers for statically scheduled accelerators. This includes generating symbolic representations of every stage in the application compiler and the hardware, leveraging scheduling information for automatically generating translation validation queries, and implementing performance enhancements for effective formal verification. This work provides a blueprint for rigorous verification of compilers and generators for hardware accelerators.

I. INTRODUCTION

With the slowdown in technology scaling, domain-specific hardware accelerators have become key to improving the performance and efficiency of compute-intensive applications. Recent research explores programmable accelerator generators, with accompanying compilers providing the means for designing these systems productively. These accelerators, while diverse (e.g., [21], [31], [37], [36], [27], [14], [16]), all use complex multistage processes to compile code from high-level languages down to accelerator instructions. Verification of these systems typically relies on comparing “golden” outputs from CPU implementations against accelerator outputs in register-transfer level (RTL) simulation, which is a slow and incomplete process for nontrivial systems. Bug root cause analysis is also challenging due to the numerous compiler stages involved.

Formal translation validation [32], [29], [38] offers a more rigorous approach. While prior work exists for software compilers (e.g., GCC [29], Halide [11]) and high-level synthesis (HLS) [23], [22], [20], ours is the first work to apply translation validation to the complex software-hardware compilation flow found in accelerator application compilers. Importantly, our work requires tackling new challenges, including dealing with many heterogeneous intermediate representations and handling notions of equivalence modulo scheduling differences.

In this paper, we present a full case study demonstrating an automated formal translation validation methodology for a statically scheduled accelerator compiler targeting a coarse-grained reconfigurable array [21]. Our approach provides guarantees against potential bugs introduced by the compiler

and also enables quick localization of any detected bugs to a specific stage in the compiler. Additionally, as our system uses a formal representation of the accelerator hardware together with various intermediate representations (IRs) in the same semantic framework, both the hardware and software compiler results are verified simultaneously.

To treat the heterogeneous multistage pipeline uniformly, we encode everything, including the application, the compiler’s IR, and the hardware RTL design, as transition systems represented using satisfiability modulo theories (SMT) formulas. This translation is automated and can adapt to both hardware and compiler changes. We define what it means for two representations to be equivalent by taking into account scheduling information present in the application compiler and then use this definition to check the equivalence of successive stages using an SMT-based model checker. Finally, we extend our technique with *symbolic starting states*, which vastly improves the performance of the system.

The contributions of our case study include:

- A uniform SMT-based formal representation for every intermediate stage in the compiler, encompassing both hardware and software, including the front-end application code, the dataflow IR, and the RTL hardware design.
- A methodology for formal verification via translation validation across every major stage of the application compiler. Our technique leverages scheduling information present in the compiler to automatically manage timing differences across stages.
- A technique for improving translation validation performance through the use of symbolic starting states with automatically generated state constraints.

We evaluate our translation validation system on 20 applications with a wide range of complexity. We verify every stage of the application compiler for these applications. The runtime overhead of these verification checks ranges from 1 minute for simple applications to 375 minutes for complex applications. The use of symbolic starting states significantly reduces memory consumption and runtime and is necessary for the verification of large applications. We introduced bugs in each stage of the compiler and confirmed that our approach can find all of them. Additionally, we found several previously unknown real bugs in our application compiler.

II. RELATED WORK

This work focuses on translation validation [32], [29], [3], a formal technique for proving the equivalence of two pieces of code, before and after translation by a compiler. Translation validation aims to prove that a particular run of a compiler pass introduces no bugs, rather than trying to prove in advance that the compiler pass will never introduce bugs. In order to apply translation validation, the code that is being validated before and after the translation needs to be represented in the same semantic framework. We leverage the language of SMT [5] for this purpose.

Translation validation has been used to verify software like Halide as it gets compiled from a high-level description into C code [32], [29], [3], hardware as it goes through optimization passes [15], and HLS systems that compile C to Verilog [23], [22], [20], [17]. However, no prior work addresses the problem of performing translation validation throughout a complex end-to-end application compiler flow targeting hardware accelerators.

HLS systems generate Verilog hardware from a C program, while the application compilers targeted in this work compile applications to existing hardware. This important distinction means that much of the infrastructure created for verifying HLS is not applicable to the problem addressed in this work. Translation validation systems designed for HLS cannot be directly applied to our problem for at least two reasons. First, translation validation requires a method to translate each compiler IR into symbolic expressions, so systems based on other IRs cannot be applied directly. Second, the definition of equality in this work differs significantly from the definition of equality in prior work. In particular, we use dataflow graphs as intermediate representations in our application compiler, while prior works primarily use control flow graphs or custom representations. Each of these has different definitions of equality and therefore different validation infrastructure.

There have also been prior attempts at verifying the hardware and software of an application compiler, but these attempts either do not provide the same benefits as translation validation or do not provide formal guarantees. For example, [10] uses hardware/software co-verification to functionally verify an application as it gets mapped to hardware. However, it uses only simulation-based verification, not formal methods. ILA [18], [19], a formal software/hardware interface for accelerators, can be used to verify an instruction-level specification of an accelerator with respect to the hardware implementation. While ILA also can be used to do complex, system-level verification of a hardware/software system, its aims are different from those of this project. ILA can be used to formally verify that the specification of an instruction matches the hardware implementation, while this work aims to verify that each stage of an application compiler produces formally correct results. One noteworthy difference is that translation validation across every major compilation stage has the benefit of enabling much easier bug localization.

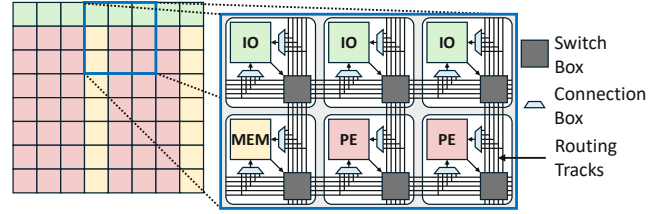


Fig. 1. Architecture of a CGRA with three types of tiles that communicate through a reconfigurable interconnect: IO tiles that pass data to and from the array, MEM tiles that buffer data, and PE tiles that do arithmetic operations.

III. BACKGROUND

A. CGRA Architecture and Application Compiler

Our translation validation case study is carried out in the context of an open-source application compiler for coarse-grained reconfigurable array (CGRA)-based programmable accelerators [21], [1]. The implementation of our verification tool is also open-source [2]. We use an SMT-based model checker for verification. We briefly describe these components to provide context.

CGRAs are a class of programmable accelerators composed of a grid of tiles, as shown in Fig. 1. CGRAs have different types of tiles: processing element (PE) tiles, memory (MEM) tiles, and input/output (IO) tiles. PE tiles perform computations, MEM tiles buffer data, and IO tiles send data to and from the grid of tiles. These tiles communicate through a reconfigurable interconnect composed of horizontal and vertical routing tracks, switch boxes (SB), and connection boxes (CB). While many types of CGRA interconnects exist [28], the one we consider in this work is similar to FPGA interconnects in that it is statically configured. This means that, during the execution of the application, the configured routing connections act as wires. Any tile can send data to any other tile in a single cycle. Additionally, there are configurable pipelining registers in every switch box, allowing multi-cycle connections between tiles.

We use the custom CGRA application compiler illustrated in Fig. 2 and presented in detail in [28]. This compiler encompasses application specification, scheduling, compute mapping, memory mapping, place and route, pipelining, and bitstream generation [21].

B. SMT-based Model Checking

SMT is the problem of determining the satisfiability of a first-order logic formula with respect to a given theory. Introductions to SMT can be found in [6], [7]. In this paper, we assume a fixed background theory $\mathcal{T}$ and assume that all formulas are interpreted according to $\mathcal{T}$. In practice, we primarily rely on the theory of bit-vectors, which includes operators corresponding to primitives in our various IRs.

A *symbolic transition system* (STS) is a tuple $\mathcal{S} := \langle X, I, T \rangle$, where X is a finite set of state variables, $I(X)$ is an SMT formula denoting the initial states of the system, and $T(X, X')$ is an SMT formula expressing a transition relation. Here, X' is the set obtained by replacing each variable $x \in X$ with a variable x' of the same sort (x' is the variable for the

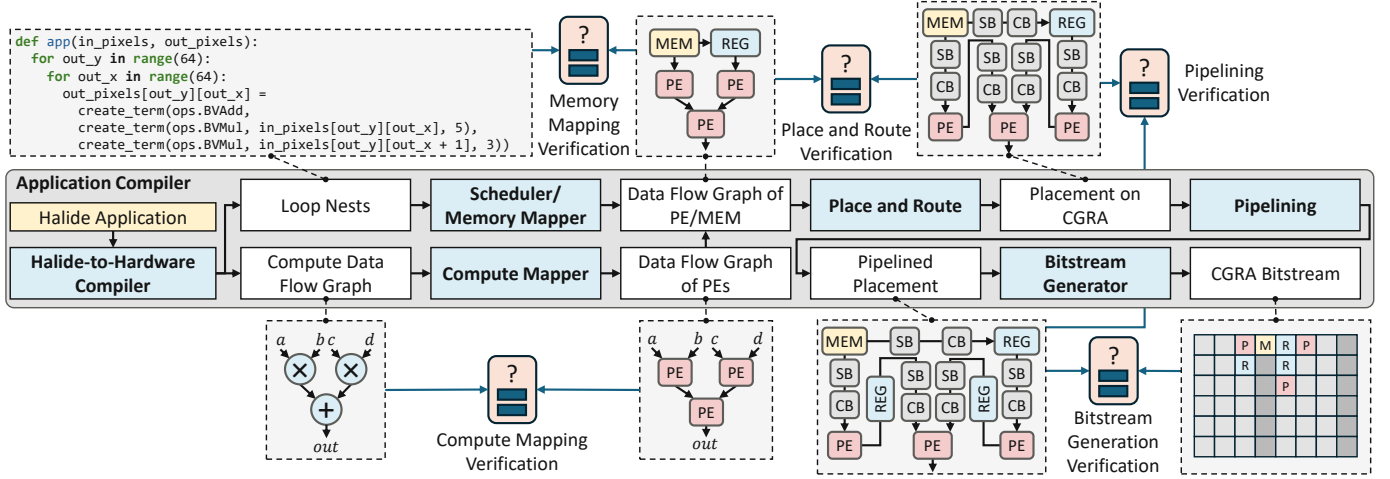


Fig. 2. CGRA application compiler encompassing Halide compilation, compute mapping, scheduling and memory mapping, place and route, pipelining, and bitstream generation. The orange boxes represent the translation validation checks we introduced.

next state of x). It is often convenient to have multiple copies of the state variables X . We use $X@n$ to denote the set of variables obtained by replacing each variable $x \in X$ with a new variable called $x@n$ of the same sort.

Bounded model checking (BMC) is a bug-finding technique which attempts to find a counterexample of length k for a proposed invariant property, $P(X)$, of some STS [8]. A single BMC query at bound k for an invariant property uses an SMT solver to check the satisfiability of the following formula: $I(X@0) \wedge (\bigwedge_{i=0}^{k-1} T(X@i, X@(i+1))) \wedge \neg P(X@k)$. If the query is satisfiable, there is a bug.

IV. SYMBOLIC REPRESENTATIONS IN SMT

To perform SMT-based translation validation for a specific compiler stage, both the input to and output from the compiler stage need to be representable as SMT terms or SMT-based symbolic transition systems. In this section, we describe our first contribution, which is to show how this can be done for each stage, from the high-level Halide specification down to the configured CGRA. We provide examples for each stage of the compiler in the appendix.

A. Application Specification

First, we need a representation of the application specification. This is written in Halide and compiled through the Halide-to-Hardware compiler [24] to generate the compute kernels in the application and the loop nests that specify how data is streamed through these kernels. These are consumed by different parts of the compiler, so we need symbolic representations for both.

Compute Kernels: CoreIR [12] is an LLVM-style hardware compiler and IR. The Halide-to-Hardware compiler uses it as the target for compute kernels, which contain dataflow graphs consisting only of arithmetic operations, e.g., add, sub, mul, shift, etc. When designing CoreIR, the semantics of these operations were based on the SMT-LIB standard [4]. One benefit of this decision is realized here, as it makes the translation of each node in the CoreIR graph to its symbolic representation

straightforward. A compute kernel can be translated into a symbolic representation by traversing the dataflow graph and constructing the SMT term corresponding to each node. Edges in the dataflow graph are transformed into equality statements. An AST-like visualization of the symbolic representation of a simple sum-of-products compute kernel for a convolution application is shown in the bottom left corner of Fig. 2.

Loop Nests: We create a new code generation target for Halide to formally represent the loop nests. We generate Python code to create the formal representation. This has the benefit of enabling better metaprogramming and integration with other compiler infrastructure, but requires an SMT solver API in a high-level language. We use Pono [25], an SMT-based model checker which provides an API for creating and reasoning about symbolic transition systems. Pono is built on top of smt-switch [26], a tool that makes it easy to use a variety of different SMT solvers as back ends. An example of the generated code for the convolution application conv_1_2 is shown in the top left corner of Fig. 2.

B. Compute-Mapped Dataflow Graph of PEs

The compute mapper replaces the CoreIR operations in the compute kernels with PE tiles configured to perform the same operations. To represent the compute-mapped graph symbolically, we need a symbolic representation of the processing elements. As described in [21], the PEs in our target CGRA are specified in PEak [13], a domain-specific language for processing elements. PEak provides direct support for interpreting a program as an SMT-based STS. We leverage the formal interpretation of PEak to symbolically represent the compute-mapped dataflow graphs. We handle any state that may be present in the PEs by creating new state variables in the STS. An AST-like visualization of a symbolic representation of the compute-mapped dataflow graph is shown in the bottom middle of Fig. 2. Not shown is that each “PE” block is configured to perform an arithmetic operation. These blocks are translated to SMT terms using PEak’s ability to provide formal models.

C. Fully-Mapped Dataflow Graph of PEs and MEMs

After applying the memory-mapping stage to the loop nests, we can construct a fully-mapped dataflow graph which contains both configured PE tiles and memory tiles. The memory tiles are specified in Lake [21], a Python-embedded language for specifying streaming memories. Unlike CoreIR and PEak, Lake has no direct support for generating SMT. Thus, to represent these memories in SMT, we first generate Verilog and then translate the Verilog to SMT-based symbolic transition systems using Yosys [34], an open-source synthesis tool that can consume Verilog and produce SMT.

To obtain the symbolic representation for the fully-mapped dataflow graph, we traverse each node of the graph in topological order, substituting in the corresponding SMT representation, either the one generated by PEak or the one from the Yosys SMT flow. A visualization of a fully-mapped dataflow graph is shown in the top middle of Fig. 2.

D. Place and Route and Pipelining

The next stage of the application compiler is place and route (PnR). The PnR tool adds new nodes to the dataflow graph for routing data between the tiles. The types of nodes introduced in PnR are SB nodes for switch boxes, CB nodes for connection boxes, RMUX nodes for register multiplexers, and REG nodes for pipelining registers.

Each of the PnR nodes has a straightforward symbolic representation as an SMT expression. The configurable interconnect is made up of multiplexers, and configured multiplexers are essentially just wires. Therefore, SB, CB, and RMUX nodes can be represented as equalities (relating the nodes at either end) in SMT. REG nodes just need to delay the input to the node by one clock cycle and can be implemented with a simple modification of the STS. The PnR graph is translated into SMT in the same manner as the fully-mapped dataflow graph.

The symbolic representation of the pipelined placement result is generated using the same method as the PnR graph.

E. CGRA Verilog

The final stage of application compilation is bitstream generation. To symbolically represent the bitstream-configured CGRA hardware, we must represent the CGRA Verilog as SMT terms. CGRAs, like FPGAs, have a reconfigurable interconnect which, in its unconfigured state, contains combinational loops. Yosys cannot translate hardware circuits with combinational loops. However, the compiler guarantees that in a correctly configured application, there will be no combinational loops. Therefore, we first apply the configuration to the CGRA RTL design before using Yosys.

The flow for generating an SMT-based STS representation of the configured CGRA is as follows: load the CGRA Verilog into Yosys, use Yosys to flatten and simplify the design, output the flattened design to a new Verilog file, replace all configuration registers within the flattened design with constant values based on the bitstream, load the modified Verilog back into Yosys, use Yosys to propagate the configuration register values and simplify as much as possible, and finally, use

Yosys to generate an SMT-based STS. Note that configuration bitstreams are composed of address-data pairs, where the address uniquely identifies a configuration register in the design. We can thus automatically identify which registers (in the flattened Verilog) to replace with constant values by leveraging the compiler's bitstream generation infrastructure to analyze the address and map it to a specific Verilog register.

V. TRANSLATION VALIDATION

After symbolically representing the application at each stage, we can construct the corresponding translation validation queries at each stage. Each query relies on two symbolic representations that may be different in structure. In this section, we describe how the queries are created for each stage.

For simplicity, we assign the following names to the translation validation checks, as shown in Fig. 2. *Compute mapping verification* is between the compute dataflow graph and the dataflow graph of PEs. *Memory mapping verification* is between loop nests and the dataflow graph of PEs and MEMs. *Place and route verification* is between the dataflow graph of PEs and MEMs and the PnR result. *Pipelining verification* is between the PnR result and the pipelined PnR result. *Bitstream generation verification* is between the pipelined PnR result and the CGRA bitstream.

A. Compute Mapping Verification

We verify that the compute dataflow graph and the compute-mapped dataflow graph of PEs are equivalent. Here, equivalence means that the graphs have the same outputs given the same inputs, with a timing delay that is discussed later in the section.

First, we identify I_c , the set of inputs to the compute dataflow graph, and O_c , the set of outputs from it. Every input i in I_c has a corresponding input i_m , representing the input to the mapped graph, and likewise every output o in O_c has a corresponding output o_m , representing the output from the mapped graph.

We check equivalence using bounded model checking with an input constraint. The input constraint gets added to the BMC formula at each unroll. The input constraint is simply: $\forall i \in I_c. i = i_m$. The property P to check is: $\exists o \in O_c. o \neq o_m$. If this check is satisfiable, then our two dataflow graphs are not equivalent and there is a bug. Although the compute dataflow graph has no state, the compute-mapped dataflow graph does, as PEs may have pipelining registers. The STS for the mapped dataflow graph uses state variables to model the sequential behavior of the PE hardware. We do an analysis of the mapped dataflow graph to determine how many cycles are needed to propagate from the inputs to the outputs and use that bound in our BMC check.

B. Memory Mapping Verification

We verify that the loop nests compiled from Halide are equivalent to the mapped dataflow graph of PEs and memory tiles. Equivalence means that, given an input array of pixels that are fed to the loop nests, if you stream the pixels into the

dataflow graph of PEs and memory tiles, the stream of pixels produced some time later is equivalent to the output array of pixels from the loop nests.

Compared to the compute mapping verification described in Section V-A, memory mapping verification is more challenging because the two sides of the validation check are structurally very different. For the loop nests, the input is a multidimensional array of pixels and the computation to get the output pixels is untimed. For the mapped dataflow graph of PEs and MEMs, the input is a stream of pixels in time, where the dataflow graph expects a different pixel every cycle, and the computation to produce the output stream has memories and pipelining registers.

To address this challenge, we add constraints to the STS to ensure that the inputs of the two sides of the verification check are equivalent. We flatten the multidimensional array of input pixels to the loop nests into a one-dimensional array. The flattening is based on the order in which the pixels are sent to the mapped data flow graph, which, importantly, is explicitly specified in the Halide application schedule. Next, we create a look-up table (LUT) within the STS that uses the array index as the LUT index and the symbol representing the pixel as the value. After creating this LUT, we constrain the input of the dataflow graph to be equal to the output of the LUT, using the current cycle as the input index. This models the simple behavior of sending one input pixel into an input of the dataflow graph each cycle.

If the pattern of input pixels is different, a more complex constraint is needed. For example, within our compiler, we have the ability to unroll each input stream into the dataflow graph. If we unroll a mapped dataflow graph input by 2, on cycle 0, pixel 0 will be sent to dataflow graph input 0, and pixel 1 will be sent to dataflow graph input 1; then, on cycle 1, pixel 2 will be sent to dataflow graph input 0, and pixel 3 will be sent to dataflow graph input 1, and so on.

Let I_h, O_h be the sets of multidimensional arrays corresponding to the inputs and outputs of the loop nests. Each array in I_h has u corresponding symbols in the mapped dataflow graph, where u is the input unrolling factor. If each array in I_h is i , then the u corresponding symbols in the mapped dataflow graph are $i_{m,0}, i_{m,1}, \dots, i_{m,u-1}$.

Similarly, each array in O_h has v corresponding symbols in the mapped dataflow graph, where v is the output unrolling factor. If each array in O_h is o , then the v corresponding symbols in the mapped dataflow graph are $o_{m,0}, o_{m,1}, \dots, o_{m,v-1}$. The symbols $i_{m,j}$ and $o_{m,j}$ are bit-vectors representing the input and output signals of the mapped application.

The input constraint over the loop nest inputs from I_h is $\forall j \in [0, u). \forall i \in I_h. i[c \cdot u + j] = i_{m,j}^c$, where u is the input unrolling factor, and c is the current cycle.

After constraining the inputs for this verification check, we can create the output verification term. We similarly flatten the loop nest's output multidimensional array into a one-dimensional array and create a LUT that uses the current cycle as the index and the pixel at that index as the output. The property that we check is: $\forall j \in [0, v). \exists o \in O_h. o[c \cdot v + j] \neq$

$o_{m,j}^{c+l}$, where l is the cycle offset determined during application scheduling.

Because applications may have pipelining registers, PEs with registers, and memories, output pixels do not immediately get produced, but get produced with a certain delay. Our key insight is that for programmable accelerators that are fully statically scheduled like the CGRAs we target, this delay is determined at compile time based on the scheduling decisions made by the compiler, and we can leverage this information from the compiler to aid translation validation.

C. Place and Route Verification

In the previous subsection, the inputs and outputs of the application on either side of the compiler stage were structurally different, while in this stage they are similar. I_m and O_m are again sets of bit-vector symbols that are inputs and outputs of the dataflow graph of PEs and MEMs. Each i and o in I_m and O_m have corresponding inputs and outputs in the PnR result, which we denote i_p and o_p , respectively.

Our input constraint is $\forall i \in I_m. i^c = i_p^c$, and the output property is $\exists o \in O_m. o^c \neq o_p^{c+l}$, where again c is the current cycle, and l is the offset in cycles between the valid output pixels from the mapped dataflow graph of PEs and MEMs and the valid output pixels from the PnR dataflow graph. Again, this offset is determined during the compilation of the application. The delay comes from hardware elements like IO tiles that have pipelining registers. These tiles get added into the application during PnR, and the delay is accounted for in the application compiler.

D. Pipelining Verification

The PnR and pipelined PnR results are also structurally similar. The input constraint and output property are therefore the same as in Section V-C, with new variables for the inputs and outputs of the pipelined version of the application. During pipelining, the compiler adds pipelining registers to the PnR result. This changes the cycle count for when outputs are produced. These additional pipelining registers are accounted for during the rescheduling stage of pipelining, and we use that scheduling information to calculate any additional timing information needed.

E. Bitstream Generation Verification

In the final stage, we have symbolic representations that are structurally very different: the pipelined PnR result is a dataflow graph translated into a symbolic representation, and the bitstream-configured CGRA is the hardware of the accelerator translated into a symbolic representation. However, the inputs and outputs of the two representations are very similar. Thus, we can again use the input constraint and output property from Section V-C, with new variables for the inputs and outputs of the configured RTL design.

Our verification framework relies on several assumptions. Most importantly, it assumes that the Halide compiler, Yosys, and the SMT solver produce correct results. It also assumes that the translations from compiler IR to SMT described in

Section IV are correct. Given these assumptions, our approach will find any bug in the application compiler that manifests in incorrect outputs and can identify the stage of the compiler that introduced that bug. Our approach verifies that the entire application executes correctly, provided we use a BMC bound sufficiently large to ensure that all outputs are produced.

VI. SYMBOLIC STARTING STATES WITH AUTOMATIC CONSTRAINT GENERATION

To verify a full application with the queries described in the previous section, we need to run BMC for thousands of cycles. For example, an application that consumes 1 pixel per cycle and produces 1 output pixel per cycle needs to be run for at least 4096 cycles for an input/output image tile size of 64×64 .

This requirement only holds for applications that start from a reset state. Resetting all state ensures that we start the verification process in a valid state. To improve performance, we enable BMC to start from a symbolic starting state, allowing verification from any point in the application’s execution. This allows for parallelization by running multiple BMC instances from different starting points.

With symbolic starting states, any state in the design can start the verification check with any value. This allows the hardware to start in invalid states, and will cause memory elements that need synchronization, like the address generators in the memory tiles, to be out of sync. The address generators present within the statically scheduled memories need several constraints to work properly. First, they have a cycle counter that is reset to 0 and counts up every cycle for synchronization purposes. In a symbolic starting state, we want these registers to start at a symbolic value, and we want that value to be identical across tiles. We add a constraint that says that all of the cycle counters start at the same symbolic value. Additionally, there are address and schedule generator counters that consume the configuration of the memory tile and count in specific patterns according to the unified buffer abstraction described in [24]. For a given cycle within the application execution, the address counter, schedule counter, and dimension counter within the memory tile have one valid value. This value can be determined before the BMC property is created, and we can constrain the counters appropriately.

Given the set of state variables that are minimally required to constrain a valid starting state, constraint generation is automated and can adapt to any hardware design. In this domain, we can generate complete sets of constraints at each timestep using compiler-derived schedules. Our use of these schedules is not black-box: we rely on the precise, static knowledge of dataflow and execution timing these tools provide to extract exact invariants. The resulting symbolic constraints successfully rule out unreachable states and prevent false counterexamples in all of our evaluated applications.

This idea is similar to automatically generated inductive invariants [9], [35]. This technique allows for parallelization of the BMC check; its effects are demonstrated in the experimental evaluation section. Additionally, starting in a

TABLE I
BENCHMARKS WITH NUMBER OF PES, MEMS, AND OUTPUT LATENCY.

Small Test	#PE	#MEM	Latency	Application	#PE	#MEM	Latency
multiply	1	1	1	conv_1_2	3	1	3
arithmetic	4	1	4	conv_2_1	2	2	65
absolute	4	1	4	conv_3_3	9	2	138
boolean	5	1	3	fast corner	20	2	208
equal	4	1	3	pyramid	15	4	464
ternary	3	1	3	gaussian	12	2	141
compare	8	1	5	harris	63	6	415
ucomp	8	1	5	unsharp	57	12	429
minmax	4	1	2				
uminmax	4	1	2				
shift	4	1	4				
ushift	3	1	3				

symbolic starting state enables more thorough verification. If any unconstrained register in the design has a value that could introduce a bug into the application, symbolic starting states ensure that we find that value. If any value in any register or memory could cause a data dependency bug, symbolic starting states would find that bug as well.

VII. EXPERIMENTAL EVALUATION

We evaluate the impact of the performance improvement from Section VI, the runtime of each verification check, and coverage of bugs in the application compiler. For these experiments we use the applications listed in Table I. The 12 applications on the left are small tests, while the applications on the right are larger workloads from [33]. For each application, the number of PEs and MEMs represent the number of resources utilized by each benchmark. We use input image tiles of size 64×64 with 16 bit values and map every application to a fixed CGRA architecture with 16 rows and 32 columns of tiles. We use the model checker Pono [25] with the Boolector backend [30]. We run every BMC check using 32 cores of a 2.5 GHz Intel CPU with 252 GB of memory.

A. Symbolic Starting States

The symbolic starting state optimization described in Section VI impacts all of the stages of verification except for compute mapping verification, as that does not have any memory tiles and very little state. This optimization enables parallel processing of the verification check, as we can start many different threads at different starting points.

Without symbolic starting states, memory mapping, place and route, pipelining, and bitstream verification cannot finish even with the simplest application. The memory cost of the verification check exceeds the total system memory of 252 GB. By using symbolic starting states to parallelize the BMC check, we can verify full applications. This optimization helps mitigate the state explosion associated with unrolling the BMC check for thousands of cycles.

B. Runtime of Translation Validation

We evaluate the runtime of each of the translation validation checks on applications that do not have any bugs. For each check, we report the runtime of the application compiler without verification, the construction of the symbolic representation, and the SMT solving.

Compute Mapping Verification Runtime (min)

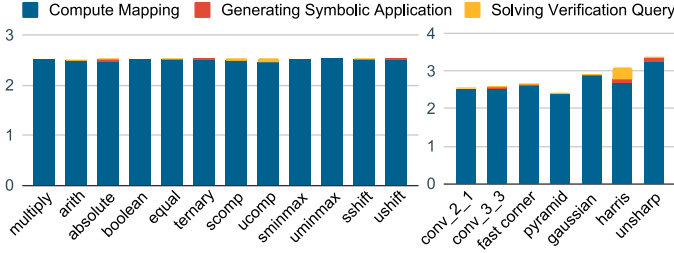


Fig. 3. Runtime of compute mapping verification, broken down into application compiler runtime without any verification, time for constructing the symbolic representation, and SMT solving time.

Memory Mapping Verification Runtime (min)

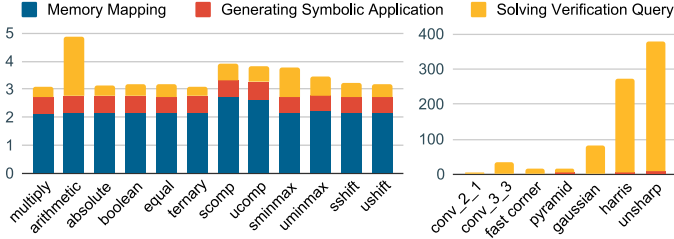


Fig. 4. Runtime of memory mapping verification.

1) *Compute Mapping Verification*: The runtime of compute mapping verification is shown in Fig 3. Overall, as both the compute dataflow graph and mapped dataflow graph of PEs do not buffer data in memory elements, the verification check is very fast, as we do not need to run the BMC algorithm for many steps. The representation of both the input to and the output from the compute mapper can be translated into SMT very quickly, as we can leverage the PEak interpretation of each node. The runtime of compute mapping verification ranges from 1 to 20 seconds.

2) *Memory Mapping Verification*: The runtime of memory mapping verification is shown in Fig. 4. As discussed in Section V-B, the two sides of this validation check are structurally very different. Therefore, the validation check is much slower than compute mapping verification, particularly for the larger application-level tests. This verification check requires running BMC for the number of cycles required to produce every output pixel.

In contrast to compute mapping verification, this verification stage is very complex, both due to the highly temporal nature of this stage and the representation of the hardware. Memory tiles are more complex than PE tiles, requiring finite state machines and counters for implementing memory addressing and access patterns. Representing memory elements in SMT using the flow described in Section IV-C results in complex representations that take more time to verify.

Overall, for smaller applications, the overhead of the verification ranges from 1 to 3 minutes, and for larger applications it ranges from 2 to 375 minutes.

3) *Place and Route Verification*: Place and route verification runtime is shown in Fig. 5. The source and target for this translation validation query are more similar than those

Place and Route Verification Runtime (min)

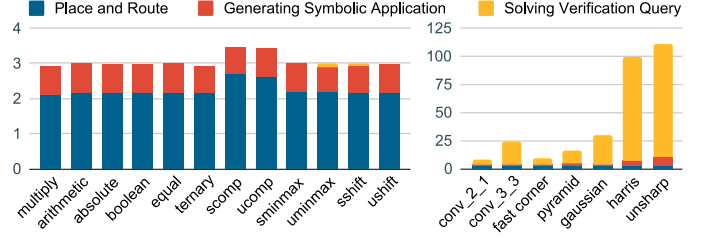


Fig. 5. Runtime of place and route verification.

Pipelining Verification Runtime (min)

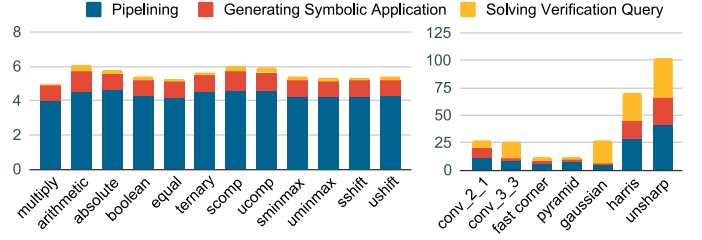


Fig. 6. Runtime of pipelining verification.

associated with memory mapping. Therefore, the verification runtime is shorter. For the small tests, running the application compiler and generating the symbolic representations take more time than the verification check itself. The overhead of this stage of verification ranges from about 1 minute for small applications to anywhere from 1 minute to 108 minutes for larger applications.

4) *Pipelining Verification*: Pipelining verification runtime is shown in Fig. 6. Like place and route verification, the representations of the application before and after pipelining are similar in structure. Pipelining will add pipelining registers to the application and reschedule the memory tiles, changing their configurations. Pipelining is a longer stage in the application compiler as it requires running PnR multiple times to find solutions with short critical paths. The overhead of pipelining verification is 1 minute for small applications, and ranges from 1 minute to 61 minutes for large applications.

5) *Bitstream Generation Verification*: Finally, the runtime of bitstream verification is shown in Fig. 7. The runtime of this verification stage is dominated by generating the symbolic application rather than the application compiler or the SMT solver. This is due to the procedure described in Section IV-E. Generating and simplifying the flattened Verilog for the accelerator using Yosys takes a long time, especially for larger applications. Larger applications also require a larger CGRA, which dramatically increases the runtime of Yosys. Overall, bitstream generation verification introduces an overhead of 8 to 9 minutes for small applications and 9 to 159 minutes for large applications.

C. Bug Coverage

1) *Introducing Bugs into the Compiler*: We report the runtime of finding various bugs in “fast corner,” a moderately complex application in our benchmark suite. First, we intro-

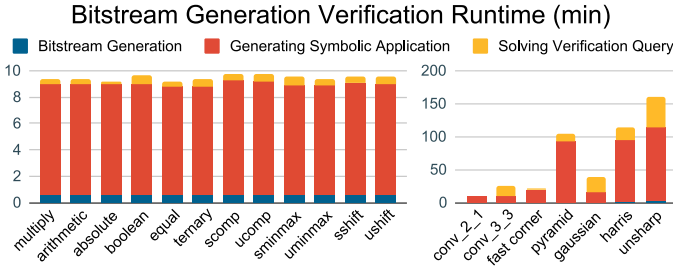


Fig. 7. Runtime of bitstream generation verification.

duce a bug in the compute mapping stage. We modify the mapping result to map a less-than operation to a PE that uses a less-than-or-equal-to operation. This bug is found 49 seconds. Next, we introduce a bug in the PE hardware where, in a three-input addition operation, the carry-in bit to the second adder is stuck at 0. This bug is found 35 seconds. Finally, we introduce a bug into the compiler where it assumes that each PE has a delay of 0 cycles when the PE takes 1 to produce an output. This bug is found 81 seconds. This stage of the compiler without verification takes 3 minutes.

Next, we discuss the bugs that are found during memory mapping verification. First, we introduce a bug in scheduling, adjusting the static schedule of a memory tile to start counting 1 cycle early. This bug is found in 3 minutes and 12 seconds. Next, we introduce a hardware bug into the memory tiles. This bug shrinks the bit width of one of the address generation counters from 5 bits to 4 bits, leading to the counter overflowing. This bug is found in 3 minutes. This stage of the application compiler without verification takes 2 minutes.

Next, we examine bugs that are found during place and route verification. First, we introduce a bug where a PE is connected to another PE through a connection that is supposed to be combinational communication but instead has a pipelining register, so it takes one cycle. This bug is found 5 minutes and 50 seconds. Next, we introduce a hardware bug into the interconnect where the configuration of a connection box is faulty and routes data to the wrong track. This bug is found 5 minutes and 55 seconds. This stage of the application compiler without verification takes 2 minutes.

Next, we examine the types of bugs that are found during pipelining verification. First, we introduce a bug where a pipelining register is added to the place and route result, but not accounted for in the scheduling of the application. This bug is found 7 minutes and 11 seconds. Next, we introduce a bug into the hardware where a single pipelining register is misconfigured and is bypassed when it should not be. This bug is found 7 minutes and 6 seconds. This stage of the application compiler without verification takes 6.5 minutes.

Finally, we examine the bugs that are found during bitstream generation verification. First, we add a bug to the compiler where a single bit of the bitstream is flipped. This bug is found in 171 minutes. Next, we introduce a bug in the accelerator hardware that shrinks the bit width of a single configuration register in the interconnect from 16 to 15 bits. This bug is found in 19 minutes. This stage of the application compiler

without verification takes 1 minute.

Without our verification system, these bugs would only be found during the Verilog simulation of the entire application. Our approach leads to much more thorough verification of the application compiler than traditional simulation. Bugs that are exercised by specific input values can easily be missed in simulation but will not be missed in our approach. Compiler passes that introduce bugs that are later masked can be missed with end-to-end simulation tests, while our approach finds bugs introduced by any stage. Simulation requires starting in a reset state, while our symbolic starting states with automated state constraints can start verification at any point in the application execution, so bugs that show up only late in the application execution may be found much faster with our approach.

2) *Bugs Found in the Application Compiler:* We discovered several real bugs in our application compiler that were previously unknown. Our application compiler has been developed for over six years [21] and until now relied on traditional simulation-based verification methods.

The first bug that we uncovered was in the PEs. There is an operation in the PE that is a multiply that takes in two 16-bit operands and returns the middle 16 bits of the 32-bit result. The bug that was found was a mismatch between the specification of the operation and the implementation within the PE. The operation was specified as two 16-bit operands that were zero padded to 32 bits, multiplied, and shifted right by 8 bits. The bottom 16 bits of the result were to be returned. This specification did not match the hardware; instead, the first step should have been to sign extend the two 16-bit operands to 32 bits. During compute mapping, this bug was found in 1.5 minutes.

The second bug was a hardware bug related to configuration register placement. The memory tiles in the CGRA can be configured to act as ROMs. In ROM mode, there is a write enable signal in the memory tile that should be unused during application execution. However, this signal was left enabled, and a configuration register present in a connection box was used to make sure no signal was routed to this input. This was a temporary fix that was forgotten about and never permanently addressed. In the memory mapping verification stage, this bug was found in 4 minutes and 12 seconds.

Finally, we discovered several scheduling bugs related to pipelining. When rescheduling an application after pipelining, any latency that is not accounted for may cause incorrect outputs. Identifying when the pipelining stage of the compiler causes the Verilog simulation to fail is difficult and often requires a significant amount of debugging. Our validation check found 5 rescheduling bugs in the rescheduling stage of pipelining in seconds for smaller applications.

VIII. CONCLUSION

In this work, we presented a methodology for automated translation validation of an application compiler for statically scheduled accelerators. We introduced an SMT-based formal representation for every stage in the compiler, a way to

perform translation validation of each of the stages leveraging scheduling information in the compiler, and a technique for dramatically improving performance through symbolic starting states. This is the first work that we are aware of to apply translation validation for a complex accelerator application compiler that combines both hardware and software representations. Our work enables rigorous verification and much better bug localization than traditional simulation-based approaches.

IX. ACKNOWLEDGMENTS

This work was supported by funding from the SRC JUMP 2.0 PRISM Center, an NSF CAREER Award (number: 2238006), a research gift from Cisco, an Apple Stanford EE PhD Fellowship in Integrated Systems, and the Stanford Agile Hardware Center.

REFERENCES

- [1] AHA Application Compiler. <https://github.com/StanfordAHA/aha>. Accessed: 2025-03-01.
- [2] Verification Implementation. https://github.com/jack-melchert/verified_agile_hardware. Accessed: 2025-03-01.
- [3] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A Translation Validator for Optimizing Compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 291–295. Springer Berlin Heidelberg, 2005.
- [4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [5] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of Model Checking*, pages 305–343, 2018.
- [6] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018.
- [7] Clark Barrett, Cesare Tinelli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, and Yoni Zohar. Satisfiability Modulo Theories: A Beginner’s Tutorial. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Proceedings of the 26th International Symposium on Formal Methods (FM ’24), Part II*, volume 14934 of *Lecture Notes in Computer Science*, pages 571–596. Springer Nature Switzerland, September 2024. Milan, Italy.
- [8] 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. Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [9] Satrajit Chatterjee and Michael Kishinevsky. Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 321–338. Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [10] Shuo-Hung Chen, Hsiao-Mei Lin, Hsin-Wen Wei, Yi-Cheng Chen, Chih-Tsun Huang, and Yeh-Ching Chung. Hardware/Software Co-designed Accelerator for Vector Graphics Applications. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 108–114, 2011.
- [11] Basile Clément and Albert Cohen. End-to-End Translation Validation for the Halide Language. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.
- [12] Ross Daly, Leonard Truong, and Pat Hanrahan. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Workshop on Open-Source EDA Technology (WOSET)*, 2018.
- [13] Caleb Donovan, Jackson Melchert, Ross Daly, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. PEak: A Single Source of Truth for Hardware Design and Verification. *ACM Transactions on Embedded Computing Systems*, November 2024.
- [14] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564, 2022.
- [15] Evgenii I Goldberg, Mukul R Prasad, and Robert K Brayton. Using SAT for Combinational Equivalence Checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 114–121. IEEE, 2001.
- [16] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: Using Epimorphism to Map Applications on CGRAs. In *Proceedings of the 49th Annual Design Automation Conference, DAC ’12*, page 1284–1291, New York, NY, USA, 2012. Association for Computing Machinery.
- [17] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.
- [18] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. INVITED: Generalizing the ISA to the ILA: A Software/Hardware Interface for Accelerator-rich Platforms. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4, 2023.
- [19] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.*, 24(1), December 2018.
- [20] Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal, and Pramod Kumar. An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):556–569, 2008.
- [21] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovan, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torg, Fredrik Kjolstad, and Priyanka Raina. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embed. Comput. Syst.*, 22(2), January 2023.
- [22] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating High-Level Synthesis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 459–472. Springer Berlin Heidelberg, 2008.
- [23] Alan Leung, Dimitar Bounov, and Sorin Lerner. C-to-Verilog Translation Validation. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 42–47, 2015.
- [24] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Trans. Archit. Code Optim.*, 20(2), March 2023.
- [25] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark W. Barrett. Pono: A Flexible and Extensible SMT-Based Model Checker. In *International Conference on Computer Aided Verification*, 2021.
- [26] Makai Mann, Amalee Wilson, Yoni Zohar, Lindsey Stuntz, Cesare Tinelli, and Clark Barrett. Smt-Switch: A Generic C++ API for SMT Solving. <https://github.com/stanford-centaur/smt-switch>, 2024.
- [27] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pages 166–173, 2002.
- [28] Jackson Melchert, Yuchen Mei, Kalhan Koul, Qiaoyi Liu, Mark Horowitz, and Priyanka Raina. Cascade: An Application Pipelining Toolkit for Coarse-Grained Reconfigurable Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2024.
- [29] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI ’00*, page 83–94. Association for Computing Machinery, 2000.
- [30] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In Hana Chockler and Georg Weissenbacher,

editors, *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 1*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

- [31] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A General Constraint-Centric Scheduling Framework for Spatial Architectures. *SIGPLAN Not.*, 48(6):495–506, June 2013.
- [32] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer Berlin Heidelberg, 1998.
- [33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, page 519–530. Association for Computing Machinery, 2013.
- [34] Claire Wolf. Yosys Open SYnthesis Suite. <https://yosyshq.net/yosys/>.
- [35] Jiahui Xu and Lana Josipović. Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.
- [36] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *Proceedings of the 48th Annual International Symposium on Computer Architecture, ISCA '21*, page 1041–1054. IEEE Press, 2021.
- [37] Zhongyuan Zhao, Weiguang Sheng, Qin Wang, Wenzhi Yin, Pengfei Ye, Jinchao Li, and Zhigang Mao. Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2201–2219, 2020.
- [38] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design*, 27(3):335–360, November 2005.

APPENDIX

Our 20 applications generate symbolic representations whose sizes range from a few lines to roughly 200,000 lines. Here, we show excerpts from the simplest application, `conv_1_2`. The representations for this simple application range from a few lines for the compute kernel to over 4000 lines for the final CGRA Verilog.

A. Compute Kernels

```
(bvadd
  (bvadd $e3
    (bvmul #h0005 $e2))
    (bvmul #h0003 $e1))
```

The single compute kernel in `conv_1_2` has three inputs and computes the following: $(e3 + (5 * e2)) + (3 * e1)$, where 5 and 3 are the kernel weights in this application.

B. Loop Nests

This is an excerpt from a 61-line representation:

```
(let
  (($e0
    (bvadd
      (bvmul in_0_0 #h0005)
      (bvmul in_1_0 #h0003))))
  (let
    (($e1
      (bvadd
        (bvmul in_1_0 #h0005)
        (bvmul in_2_0 #h0003))))
    (let
      (($e2
```

```
(bvadd
  (bvmul in_2_0 #h0005)
  (bvmul in_3_0 #h0003))))
(let
  (($e3
    (bvadd
      (bvmul in_0_1 #h0005)
      (bvmul in_1_1 #h0003))))
  ...
```

Here, we assign each output variable a value based on the addition of the multiplication of two input pixels with kernel values.

C. Compute-Mapped Dataflow Graph of PEs

```
((_ extract 15 0)
  (bvadd
    (concat #b0
      ((_ extract 15 0)
        (bvadd
          (concat #b0
            ((_ extract 15 0)
              (bvmul #h00000005
                (concat #h0000 $e2))))
            (concat #b0 $e3))))
    (concat #b0
      ((_ extract 15 0)
        (bvmul #h00000003
          (concat
            (ite
              (= #b1
                ((_ extract 47 47) IVAR_V_415@0))
                #hfff #h000) $e1))))))
```

In this stage, the representation is generated by taking the dataflow graph of configured PEs and simplifying it to remove all unused hardware. In this application, we have two PEs performing multiply-add operations on 16-bit operands with a 32-bit multiplier.

D. Fully-Mapped Dataflow Graph of PEs and MEMs

This is an excerpt from a 1284-line representation:

```
...
(bvand
  (bvand
    (bvand clk_en$const0.next
      (bvand clk_en$const0
        (ite
          (and
            (= V_0$const0 V_0$const0_inter.next)
            (= bmc_counter.next
              (bvadd bmc_counter #h0001))
            (= V_0$const0.next V_0$const0_inter)
            (= V_1$const0 V_1$const0_inter.next)
            (= V_1$const0.next V_1$const0_inter)
            (= V_2$const0 V_2$const0_inter.next)
            (= V_2$const0.next V_2$const0_inter)
            (= V_3$const0 V_3$const0_inter.next)
            (= V_3$const0.next V_3$const0_inter)
            (= V_4$const0 V_4$const0_inter.next)
            (= V_4$const0.next V_4$const0_inter)
            (= V_5$const0 V_5$const0_inter.next)
            (= V_5$const0.next V_5$const0_inter)
            #b1 #b0)))
      (= state5.next $e2) #b1 #b0))
    (ite
      (= state7.next
        (ite
          (= #b1 flush_mem) #h0000
          (bvadd $e2
```


Unifying DQMax#SAT and DSSAT: Polynomial-Time Reduction and Applications

Ilo Chen*, Che Cheng[†] and Jie-Hong Roland Jiang*[†]

*Department of Electrical Engineering, [†]Graduate Institute of Electronics Engineering
National Taiwan University, Taipei, Taiwan 106319
Email: {b11901173, f11943097, jhjiang}@ntu.edu.tw



Abstract—DQMax#SAT is a newly developed extension of Max#SAT that incorporates Henkin-type quantifiers and is utilized in function synthesis to enhance program security. This study refutes the earlier hypothesis that efficient reduction from DQMax#SAT to the dependency stochastic Boolean satisfiability (DSSAT) is not possible. We introduce a conversion algorithm that eliminates variables from dependency sets to achieve the reduction. Experimental results show that transforming DQMax#SAT formulas that encode function synthesis applications into DSSAT formulas and solving them using an off-the-shelf DSSAT solver yields superior results compared to directly solving them with the leading DQMax#SAT solver. This research not only clarifies the relationship between DQMax#SAT and DSSAT but also equips developers with conversion tools to assist in the evaluation of both DQMax#SAT and DSSAT solvers.

I. INTRODUCTION

Current research in system synthesis, formal verification, software security, and other domains is increasingly centered on encoding complex decision problems concisely and synthesizing optimal strategies. These applications have spurred advancements in Boolean satisfiability, expanding it to include quantified Boolean formula (QBF), dependency QBF (DQBF), stochastic Boolean satisfiability (SSAT), among others [1].

The dependency quantified Max#SAT (DQMax#SAT) problem [2], which extends the Max#SAT problem [3] by allowing explicit specification of variable dependencies, has recently been formulated for function synthesis in software security applications. DQMax#SAT is shown in [2] to be NEXPTIME-complete, allowing it to encode complex problems such as quantitative robustness [4] and information leakage of programs [5]. In [2], DQMax#SAT instances are transformed into the Max#SAT instances and solved using off-the-shelf Max#SAT solvers.

On the other hand, the dependency SSAT (DSSAT) problem [6] is closely related to the DQMax#SAT problem. DSSAT extends SSAT in a way similar to that of DQBF extending QBF by allowing branching quantifiers, a.k.a. Henkin quantifiers [7], to admit explicit specification of variable dependencies. Its decision version is also NEXPTIME-complete, and its applications in probabilistic partial design and decentralized partially observable Markov decision process (Dec-POMDP) have been studied. The DSSAT preprocessor DSSATpre [8] applies preprocessing techniques and converts DSSAT formulas

into SSAT formulas, allowing them to be solved by off-the-shelf SSAT solvers.

Although DQMax#SAT and DSSAT are conceptually similar, with DQMax#SAT syntactically encompassing DSSAT (subject to a generalization with weighted counting), it was previously argued in [2] that an efficient reduction from DQMax#SAT to DSSAT is infeasible. In this work, we challenge this claim by presenting a polynomial-time reduction from DQMax#SAT to DSSAT, achieved by eliminating existential variables from dependency sets. The efficiency of such conversion is demonstrated empirically, where we transform DQMax#SAT formulas into DSSAT ones and solve them using the DSSAT preprocessor DSSATpre [8] and SSAT solver SharpSSAT. Experimental results show that this approach significantly outperforms the state-of-the-art DQMax#SAT solver provided in [2] across all benchmark formulas. We expect this work to support developers of DQMax#SAT and DSSAT solvers in assessing their tools and refining benchmarks.

II. PRELIMINARIES

In this section, we define the notations used throughout the paper and provide the required background for DSSAT and DQMax#SAT.

We denote Boolean values TRUE and FALSE by $\top$ and $\perp$, respectively, and interpret Boolean connectives $\neg$, $\vee$, $\wedge$, and $\leftrightarrow$ as negation, disjunction, conjunction, and equivalence, respectively. A *literal* l is a Boolean variable x or its negation $\neg x$. A *cube* is a conjunction of literals, and a *clause* is a disjunction of literals. We alternatively treat cubes and clauses as sets of literals whenever appropriate. We denote the set of all variables appearing in a Boolean formula ϕ as $\text{var}(\phi)$. Given a Boolean formula ϕ , the *substitution* of variable $x \in \text{var}(\phi)$ with formula ψ , denoted as $\phi[\psi/x]$, replaces all occurrences of x in ϕ by ψ . A *quantified formula* is of the form $\Phi = Q.\phi$, where Q is the *quantifier prefix* and ϕ is a quantifier-free formula called the *matrix*.

An *assignment* α over a set $V = \{v_1, \dots, v_n\}$ of Boolean variables is a mapping $V \rightarrow \{\top, \perp\}$. We alternatively treat an assignment α as a cube C such that variable x with literal x (resp., $\neg x$) appearing in C has $\alpha(x) = \top$ (resp., $\alpha(x) = \perp$). We write $\phi[\alpha]$ to denote $\phi[\alpha(v_1)/v_1] \dots [\alpha(v_n)/v_n]$. An assignment α *satisfies* ϕ , denoted as $\alpha \models \phi$, if $\phi[\alpha] = \top$. In

this case, α is called a *model* of ϕ . The model count of a formula ϕ is denoted by $\#[\phi]$.

A. DSSAT

Let $V = Y \cup X$ be a set of Boolean variables, where $Y = \{y_1, \dots, y_m\}$ is the set of *random* variables and $X = \{x_1, \dots, x_n\}$ is the set of existential variables. A DSSAT formula Ψ is of the form

$$\mathbf{H}^{p_1} y_1, \dots, \mathbf{H}^{p_m} y_m, \exists x_1(D_{x_1}), \dots, \exists x_n(D_{x_n}). \psi,$$

where $D_{x_i} \subseteq Y$ is the dependency set of the existential variable x_i , $p_i \in [0, 1]$ is a probability, and ψ is a quantifier-free formula. The symbol $\mathbf{H}^{p_i} y_i$ denotes a *random quantifier*, where the variable y_i evaluates to $\top$ with probability p_i and $\perp$ with probability $1 - p_i$. The probability $w[\alpha]$ of an assignment α over Y is defined as the product of the probability of its literals, i.e.,

$$w[\alpha] = \prod_{i: \alpha(y_i) = \top} p_i \cdot \prod_{i: \alpha(y_i) = \perp} (1 - p_i).$$

A strategy $F = (f_1, \dots, f_n)$ for Ψ is tuple of Boolean functions for the existential variables, where f_i is a Boolean function over the variable set D_{x_i} . We write $\psi|_F$ to denote the substitution of each x_i with f_i in ψ , and $\Psi|_F$ the quantified formula $\mathbf{H} R. \psi|_F$, which is essentially a weighted model counting instance. The *satisfying probability* of Ψ under a strategy F is the sum of the probability of its models, i.e.,

$$\Pr[\Psi|_F] = \sum_{\alpha \models \psi} w[\alpha].$$

The satisfying probability of Ψ is defined as the maximum satisfying probability across all strategies, i.e.,

$$\Pr[\Psi] = \max_F \Pr[\Psi|_F].$$

For simplicity, we assume in the sequel that all random quantifiers have probability $p = 0.5$. This is reasonable since any other probability can be converted to a formula with only probability 0.5 up to any given precision [9].

B. DQMax#SAT

Let $V = X \cup Y \cup Z$ be a set of Boolean variables, where $X = \{x_1, \dots, x_n\}$ is the set of *maximizing variables*, $Y = \{y_1, \dots, y_m\}$ is the set of *counting variables*, and $Z = \{z_1, \dots, z_l\}$ is the set of *existential variables*. A DQMax#SAT formula Φ is of the form

$$\max x_1(D_{x_1}), \dots, \max x_n(D_{x_n}), \#y_1, \dots, \#y_m, \exists z_1, \dots, \exists z_l. \phi \quad (1)$$

where $D_{x_i} \subseteq Y \cup Z$ is the dependency set of variable x_i .

A strategy $F = (f_1, \dots, f_n)$ for Φ is a tuple of Boolean functions for the maximizing variables, where f_i is a Boolean function over the variable set D_{x_i} . We again write $\phi|_F$ to denote the substitution of each x_i with f_i in ϕ and $\Phi|_F$ the quantified formula $\#Y, \exists Z. \phi|_F$.

The *model count* of Φ under a strategy F is the *projected model count* of $\Phi|_F$, denoted $\#[\Phi|_F]$, i.e., the number of

Algorithm 1 Existential-Free DQMax#SAT to DSSAT

Input: Existential-free DQMax#SAT formula $\Phi = \max X, \#Y. \phi$
Output: DSSAT formula Ψ

```

1:  $Q \leftarrow \emptyset$ 
2: for all variable  $v \in Y$  do                                 $\triangleright v$  is COUNTING
3:    $Q \leftarrow Q, \mathbf{H}^{0.5} v$                                  $\triangleright$  append random variable to  $Q$ 
4: for all variable  $v \in X$  do                                 $\triangleright v$  is MAXIMIZING
5:    $Q \leftarrow Q, \exists v(D_v)$                                  $\triangleright$  append existential variable to  $Q$ 
6: return  $\Psi \leftarrow Q. \phi$                                  $\triangleright$  By Prop. 1,  $\Pr[\Psi] = \#[\Phi]/2^{|Y|}$ .
```

assignments α on Y formula that can be extended to a model of $\phi|_F$ with some assignment on Z [10]. The model count of Φ is defined as the maximum model count across all strategies, i.e.,

$$\#[\Phi] = \max_F \#[\Phi|_F].$$

DQMax#SAT can be thought of as an extension of DSSAT by allowing dependency on existential variables. We will show that this increased flexibility does not increase its expressive power.

III. EQUIVALENCE BETWEEN DQMax#SAT AND DSSAT

Since both DQMax#SAT and DSSAT are NEXPTIME-complete, there exists a polynomial-time reduction between them. However, such a reduction may be overly complex and impractical. A reduction from DSSAT to DQMax#SAT is straightforward and has been given in [2]. The challenge lies in the dependency on existential variables Z unique to the DQMax#SAT formalism. To tackle this, we first identify a subclass of DQMax#SAT whose reduction to DSSAT is straightforward. Then, we demonstrate how to convert an arbitrary DQMax#SAT formula into an existential-free one while preserving its model count.

A. Transforming Existential-Free DQMax#SAT into DSSAT

We start by defining a subclass of DQMax#SAT formulas, called the *existential-free* DQMax#SAT formulas, which is the class of DQMax#SAT formulas without existential variables.

Definition 1 (Existential-free DQMax#SAT formula). A DQMax#SAT formula in the form of Eq. (1) is *existential-free* if the set Z of existential variables is empty.

We show that an existential-free DQMax#SAT formula is essentially a DSSAT formula in the following proposition and outline the corresponding conversion in Algorithm 1.

Proposition 1. . Given an existential-free DQMax#SAT formula

$$\Phi = \max x_1(D_{x_1}), \dots, \max x_n(D_{x_n}), \#y_1, \dots, \#y_m. \phi,$$

the DSSAT formula

$$\Psi = \mathbf{H}^{0.5} y_1, \dots, \mathbf{H}^{0.5} y_m, \exists x_1(D_{x_1}), \dots, \exists x_n(D_{x_n}). \phi$$

has satisfying probability

$$\Pr[\Psi] = \frac{\#[\Phi]}{2^{|Y|}}.$$

Proof. First, observe that the sets of valid strategies for Φ and Ψ coincide. In addition, since each random variable $y \in Y$ in Ψ is assigned uniformly with probability 0.5, every assignment α of Y has probability $w[\alpha] = 2^{-|Y|}$. It follows that given any strategy F , we have

$$\Pr[\Psi|_F] = \frac{1}{2^{|Y|}} \#[\phi|_F] = \frac{1}{2^{|Y|}} \#[\Phi|_F],$$

and therefore

$$\Pr[\Psi] = \max_F \Pr[\Psi|_F] = \frac{1}{2^{|Y|}} \max_F \#[\Phi|_F] = \frac{\#[\Phi]}{2^{|Y|}}.$$

□

This reduction runs in polynomial time as it is a mere rewriting of the prefix. Specifically, in Algorithm 1, both the for-loops in line 2 and line 4 take time linear in the number of variables. Therefore, the algorithm runs in $O(|X| + |Y| + k)$, where k is the formula size of ϕ .

B. Transforming DQMax#SAT into Existential-Free DQMax#SAT

We now show that any arbitrary DQMax#SAT formula can be transformed into an existential-free one while preserving its model count. Recall that under a strategy F , a DQMax#SAT formula reduces to one of projected model counting. In this case, we make the following definition of *Skolem function*.¹

Definition 2 (Skolem function). Let ϕ be a Boolean formula with $\text{var}(\phi) = Y \cup \{z\}$. A Boolean function f_z over variables Y is a *Skolem function* of z if

$$\exists z. \phi \equiv \phi[f_z/z].$$

We show that, using the syntax of DQMax#SAT, we can perform *Skolemization* on projected model counting formulas by converting existential variables into maximizing variables, as detailed in Proposition 2.

Proposition 2. *Let*

$$\Phi = \#Y, \exists Z. \phi$$

be a projected model counting formula and $z \in Z$ an existential variable. Then, Φ has the same model count as

$$\Phi' = \max z(Y), \#Y, \exists Z \setminus \{z\}. \phi.$$

Proof. For simplicity, we write ϕ' to denote the formula $\exists Z \setminus \{z\}. \phi$ over variables $Y \cup \{z\}$. To prove this, we consider the strategy $F = (f_z)$, where f_z is the Skolem function for z with respect to ϕ' defined in Definition 2. Since

$$\exists z. \phi' \equiv \phi'[f_z/z],$$

the model count $\#[\Phi] = \#[\Phi'|_F]$. It remains to show that F is the optimal strategy. For contradiction, suppose that $F' = (f'_z)$ is such that $\#[\Phi'|_{F'}] > \#[\Phi'|_F]$. Then, there must be some assignment α on Y such that $\alpha \models \phi'[f'_z/z]$ and $\alpha \not\models \phi'[f_z/z]$.

¹While the Boolean functions in the strategies of DSSAT formulas are also referred to as Skolem functions in the literature, we reserve the name specifically for the type of function defined in Definition 2 to avoid confusion.

Algorithm 2 DQMax#SAT to Existential-Free DQMax#SAT

Input: DQMax#SAT formula $\Phi = \max X \#Y \exists Z. \phi$

Output: Existential-free DQMax#SAT Φ_{EF}

```

1:  $\hat{Z} \leftarrow \emptyset$   $\triangleright$  Set of existential variable in dependency sets
2: for all variable  $v \in \phi$  do
3:   if  $v \in X$  then
4:     for all  $z \in D_v$  do
5:       if  $z \in Z$  then
6:          $\hat{Z} \leftarrow \hat{Z} \cup \{z\}$ 
7: while  $\hat{Z} \neq \emptyset$  do
    $\triangleright$  Step 1: remove EXISTENTIAL variable  $z$  in dependency set
   and transform to MAXIMIZING variable (Prop. 3)
8:   select  $z$  from  $\hat{Z}$ ;  $\hat{Z} \leftarrow \hat{Z} \setminus \{z\}$ 
9:   introduce fresh variable  $z'$ 
10:   $Y \leftarrow Y \cup \{z'\}$   $\triangleright$  create new COUNTING variable  $z'$ 
11:  for all variable  $v \in \phi$  do
12:    if  $v \in X$  and  $z \in D_v$  then
13:       $(D_v \setminus \{z\}) \cup \{z'\}$ 
14:     $\phi \leftarrow \phi \wedge (z \leftrightarrow z')$ 
15:     $Z \leftarrow Z \setminus \{z\}$ 
16:     $X \leftarrow X \cup \{z\}$ ;  $D_z \leftarrow Y$ 
17: while  $Z \neq \emptyset$  do
    $\triangleright$  Step 2: remove EXISTENTIAL variable  $z$  not in dependency
   set and transform to MAXIMIZING variable (Cor. 1)
18:   select  $z$  from  $Z$ ;  $Z \leftarrow Z \setminus \{z\}$ 
19:    $X \leftarrow X \cup \{z\}$ ;  $D_z \leftarrow Y$ 
20: return  $\Phi$ 

```

It follows that $f_z(\alpha) \neq f'_z(\alpha)$ and $\alpha \models \exists z. \phi'$, and thus by definition, $\alpha \models \phi'[f_z/z]$, which leads to a contradiction. □

As a corollary, given a DQMax#SAT formula, when an existential variable does not appear in any dependency set, it can be replaced by a maximizing variable without affecting the model count. The process directly corresponds to Step 2 in Algorithm 2.

Corollary 1. *Let Φ be a DQMax#SAT formula in the form of Eq. (1). If an existential variable $z \in Z$ does not appear in any of the dependency sets, then Φ has the same model count as*

$$\Phi' = \max X, \max z(Y), \#Y, \exists(Z \setminus \{z\}). \phi.$$

Proof. The corollary follows from Proposition 2 as the projected model counting for $\Phi|_F$ with respect to any given strategy F of the maximizing variables X of Φ has the same count as $\Phi'|_F$. □

When an existential variable does appear in a dependency set, it can still be replaced by a maximizing variable without affecting the model count, as we now show in the following proposition. The process directly corresponds to Step 1 in Algorithm 2.

Proposition 3. *Let Φ be a DQMax#SAT formula in the form of Eq. (1) and let $z \in Z$ be an existential variable. Φ has the same model count as*

$$\Phi' = \max x_1(D'_{x_1}), \dots, \max x_n(D'_{x_n}), \max z(Y), \#(Y \cup \{z'\}), \exists(Z \setminus \{z\}). \phi',$$

where

$$D'_x = \begin{cases} (D_x \setminus \{z\}) \cup \{z'\} & \text{if } z \in D_x \\ D_x & \text{otherwise} \end{cases}$$

and

$$\phi' = \phi \wedge (z \leftrightarrow z').$$

Proof. For each strategy F for Φ , let

$$\Phi|_F = \max z(Y), \#Y, \exists Z \setminus \{z\}. \phi|_F.$$

By Proposition 2, $\#[\Phi|_F] = \#[\Phi|_F']$.

Let $F = (f_{x_1}, \dots, f_{x_n})$ be a strategy for Φ and f_z a strategy for $\Phi|_F$. To ease notation, we write $\phi|_{(F, f_z)}$ to denote $\phi|_F[f_z/z]$.² We show that the strategy $F' = (f'_{x_1}, \dots, f'_{x_n}, f_z)$ for Φ' is such that $\#[\Phi|_{(F, f_z)}] = \#[\Phi'|_{F'}]$, where

$$f'_{x_i} = f_{x_i}[z'/z]$$

for $1 \leq i \leq n$, and that this mapping is a bijection.

It is easy to see that the mapping between (F, f_z) and F' is a bijection and that F' is a valid strategy for Φ' . To see that $\#[\Phi|_{(F, f_z)}] = \#[\Phi'|_{F'}]$, consider any assignment α on Y with $\alpha \models \phi|_{(F, f_z)}$. Let $\alpha_0 = \alpha \cup \{\neg z'\}$ and $\alpha_1 = \alpha \cup \{z'\}$ be the two assignments on $Y \cup \{z'\}$ consistent with α . We show that exactly one of $\{\alpha_0, \alpha_1\}$ is a model of $\phi'|_{F'}$. Note that $\phi'|_{F'} = \phi|_{F'} \wedge (f_z \leftrightarrow z')$.

W.L.O.G. assume that $f_z(\alpha) = \top$. Then, $\alpha_0 \not\models (f_z \leftrightarrow z')[\alpha]$, which implies that $\alpha_0 \not\models \phi'|_{F'}$. On the other hand, $\alpha_1 \models (f_z \leftrightarrow z')[\alpha]$. Since $\alpha_1(z') = f_z(\alpha) = \top$, we have

$$\phi|_{(F, f_z)}[\alpha] = \phi|_F[\top/z][\alpha] = \phi|_{F'}[\top/z'][\alpha] = \phi|_{F'}[\alpha_1].$$

It follows that $\alpha_1 \models \phi|_{F'}$, and thus $\alpha_1 \models \phi'|_{F'}$. $\square$

Using Corollary 1 and Proposition 3, we can transform any DQMax#SAT formula into existential-free form by converting all existential variables into maximizing variables. The entire process is sketched in Algorithm 2.

For the complexity analysis, it is easy to see that lines 8-16 in the while-loop are dominated by the for-loop (lines 11-13), which takes a single pass over all variables. For each variable, checking for membership in a dependency set takes time linear to the size of the set, which is also bounded by the number of total variables. Therefore, the for-loop requires at most $O((|X| + |Y| + |Z|)^2)$ time. Since each iteration of the while-loop removes one member from $\hat{Z}$, the loop will run a total of $|\hat{Z}|$ times, which is bounded by $|Z|$. Hence the first while-loop will run in $O(|Z|(|X| + |Y| + |Z|)^2)$. Similarly, the second while-loop takes $O(|Z|)$ time. Consequently, the entire algorithm is of complexity $O(|Z|(|X| + |Y| + |Z|)^2)$.

C. Transforming DQMax#SAT into DSSAT

We now present our theorem for transforming DQMax#SAT to DSSAT. The theorem directly corresponds to Algorithm 3, which makes use of the sub-procedures of Algorithms 1 and 2 presented before. Since all sub-procedures are polynomial and each sub-procedure is called only once, the full conversion

²Note that the occurrence of z in F is also replaced by f_z .

Algorithm 3 DQMax#SAT to DSSAT via Existential Elimination

Input: DQMax#SAT formula Φ

Output: DSSAT formula Ψ

- 1: $\Phi_{\text{EF}} \leftarrow \text{GENERALTOEXISTENTIALFREE}(\Phi)$ $\triangleright$ Alg. 2
- 2: $\Psi \leftarrow \text{EXISTENTIALFREETODSSAT}(\Phi_{\text{EF}})$ $\triangleright$ Alg. 1
- 3: **return** Ψ

$\triangleright$ By Thm. 1 and Prop. 1, $\Pr[\Psi] = \#[\Phi]/2^{|Y|+|\hat{Z}|}$.

algorithm is polynomial. More explicitly, the algorithm runs in $(|Z|(|X| + |Y| + |Z|)^2)$ time.

Theorem 1. Let Φ be a DQMax#SAT formula in the form of Eq. (1), and let $\hat{Z} \subseteq Z$ denote the set of existential variables that occur in some dependency sets. The DSSAT formula

$$\Psi = \mathfrak{M} Y, \mathfrak{M} \hat{Z}', \exists x_1(D'_{x_1}), \dots, \exists x_n(D'_{x_n}), \exists Z(Y).$$

$$\phi \wedge \bigwedge_{z_i \in \hat{Z}} (z_i \leftrightarrow z'_i)$$

has satisfying probability

$$\Pr[\Psi] = \frac{\#[\Phi]}{2^{|Y|+|\hat{Z}|}},$$

where $\hat{Z}' = \{z'_i \mid z_i \in \hat{Z}\}$ is a fresh copy of $\hat{Z}$, and

$$D'_{x_j} = (D_{x_j} \setminus \{z_i \mid z_i \in D_{x_j}\}) \cup \{z'_i \mid z_i \in D_{x_j}\}$$

for each $1 \leq j \leq n$.

Proof. Starting with Φ , we first apply Corollary 1 on the existential variables $Z \setminus \hat{Z}$, which gives

$$\Phi_1 = \max X, \max Z \setminus \hat{Z}, \#Y, \exists \hat{Z}. \phi$$

with $\#[\Phi_1] = \#[\Phi]$. Then, we apply Proposition 3 on the existential variables $\hat{Z}$ in Φ_1 to obtain

$$\Phi_2 = \max x_1(D'_{x_1}), \dots, \max x_n(D'_{x_n}), \max Z(Y), \#Y, \# \hat{Z}'. \phi \wedge \bigwedge_{z_i \in \hat{Z}} (z_i \leftrightarrow z'_i)$$

with $\#[\Phi_2] = \#[\Phi_1] = \#[\Phi]$. Finally, we apply Proposition 1 on Φ_2 to obtain Ψ with

$$\Pr[\Psi] = \frac{\#[\Phi_2]}{2^{|Y|+|\hat{Z}'|}} = \frac{\#[\Phi]}{2^{|Y|+|\hat{Z}|}}. \quad \square$$

IV. EXPERIMENTAL RESULTS

We implemented the conversion from the DQMax#SAT formula to the DSSAT formula in the C++ language.³ The obtained DSSAT formulas were then preprocessed and converted to SSAT formulas using DSSATpre [8] and finally solved using SharpSSAT [11]. We denote the solver provided in [2] as DQMax#SAT. All experiments are carried out on an Apple M2 chip with 24 GB of RAM.

We compare DSSATpre+SharpSSAT and DQMax#SAT in solving DQMax#SAT formulas. While DQMax#SAT provides

³The code and benchmarks are available at <https://github.com/NTU-ALComLab/DQMaxSharpSAT2DSSAT>.

TABLE I
PERFORMANCE COMPARISON OF DQMax#SAT AND DSSAT.

Benchmark	DQMax#SAT							DSSAT						
	X	Y	Z	#C	max D	$\sum D $	Time	Y'	X'	#C'	max D	$\sum D $	Time	
example_1	1	2	2	7	2	2	0.01 s	4	3	11	2	6	0.01 s	
example_2	2	2	2	7	1	2	0.01 s	4	4	11	2	6	0.01 s	
example_7	3	6	97	329	3	9	74.64 s	9	100	335	6	591	0.01 s	
example_8	9	3	150	487	2	9	78.15 s	3	159	487	3	459	0.04 s	
example_9	9	3	93	289	2	9	20.46 s	9	102	293	3	288	0.03 s	
example_10	9	3	114	355	2	9	93.31 s	3	123	355	3	351	0.01 s	
example_11	1	2	1	5	1	1	0.01 s	3	2	7	2	3	0.01 s	
C1	8	12	163	558	8	48	TO	20	171	574	12	2004	0.10 s	
C2	6	9	273	925	6	54	TO	15	279	937	9	2511	6.03 s	
C3	15	9	141	423	9	54	TO	18	156	441	9	1323	8.70 s	
C4	12	6	307	870	15	90	TO	21	319	894	15	1932	48.19 s	
C5	16	8	352	966	20	128	TO	28	368	1022	20	2944	TO	

Note: #C and #C' denote the number of clauses in the DQMax#SAT and the corresponding DSSAT formula, respectively; Y' and X' denote the variable set $Y \cup Z'$ and $X \cup Z$ as detailed in Theorem 1; max|D| and $\sum|D|$ denote the size of the largest dependency set and the total size of dependency sets, respectively; TO denotes timeout at 600 s.

options for exact and approximate computations, we focus on exact computation. All the benchmark formulas, namely, example_*i*, for $i = 1, 2, 7, 8, \dots, 11$, used in [2] are taken for evaluation. A detailed description of the formulas can be found in [2]. We note that example_1, example_2, and example_11 are simple DQMax#SAT formulas and example_7, ..., example_10 are bit-blasted formulas of security problems mentioned in [2]. For example, the formula of example_1 is

$$\Phi = \max x_1 \{z_1, z_2\} \cdot \#y_1 \cdot \#y_2 \cdot \exists z_1 \cdot \exists z_2 \cdot (x_1 \leftrightarrow y_1) \wedge (z_1 \leftrightarrow (y_1 \vee y_2)) \wedge (z_2 \leftrightarrow (y_1 \wedge y_2)).$$

The formula of example_7 before bit-blasting is

$$\Phi = \max x_1 \{z_1\} \cdot \#y_1 \cdot \#y_2 \cdot \exists z_1 \cdot (y_1 \leq x_1 \wedge x_1 \leq y_2) \wedge (z_1 = y_1 + y_2),$$

where each variable represents a 3-bit unsigned integer. Based on example_7, we further generated more challenging random DQMax#SAT formulas, C1 to C5, with their matrices in SMT format with varying numbers of clauses and variables, where each variable representing a 3- or 4-bit unsigned integer was then bit blasted.

Table I shows the results, where the number of clauses and each type of variable are given. The runtime of the DSSAT configuration includes the conversion time from DQMax#SAT to DSSAT (which all completed in milliseconds), the runtime of DSSATpre, and the runtime of SharpSSAT. As can be seen in Table I, a substantial runtime improvement is achieved, especially for larger benchmarks. Note that the maximum size of the dependency sets increases in cases where the DQMax#SAT formulas have counting set |Y| larger than max|D|, as can be seen in example_2 or C1. Since the number of random variables and the size of the dependency sets in the DSSAT formulas are directly related to the number of existential DQMax#SAT variables in the dependency sets, existential DQMax#SAT variables can be a good indicator of the required solving time.

The results suggest that the effectiveness of DSSATpre and the potential to develop preprocessing techniques for more effective DQMax#SAT solving.

V. CONCLUSION

In this paper, we established efficient reduction between the formalisms DSSAT and DQMax#SAT. In addition to the theoretical contributions, we implement the proposed reduction and demonstrate its practical value through empirical evaluation. Experimental results show substantial improvements in solving time when using our DSSAT-based approach, especially for larger instances. This confirms the feasibility and benefit of converting DQMax#SAT problems into DSSAT formulations for practical applications.

For future work, we plan to investigate the optimization of the reduction process for structure-aware transformations to explore the reverse reduction from DSSAT to DQMax#SAT for expressiveness gains, and to study subclasses of DQMax#SAT that might benefit from specialized solving techniques.

ACKNOWLEDGMENTS

This work was supported in part by the National Science and Technology Council of Taiwan under grant NSTC 111-2923-E-002-013-MY3 and 114-2221-E-002-183-MY3, and the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-113L900903.

REFERENCES

- [1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, vol. 336.
- [2] T. Vigouroux, M. Bozga, C. Ene, and L. Mounier, "Function synthesis for maximizing model counting," in *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 258–279.
- [3] D. J. Fremont, M. N. Rabe, and S. A. Seshia, "Maximum model counting," in *Proceedings of the AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, pp. 3885–3892.
- [4] S. Bardin and G. Girol, "A quantitative flavour of robust reachability," 2022. [Online]. Available: <https://arxiv.org/abs/2212.05244>

- [5] G. Smith, “On the foundations of quantitative information flow,” in *Foundations of Software Science and Computational Structures*, L. de Alfaro, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 288–302.
- [6] N.-Z. Lee and J.-H. R. Jiang, “Dependency stochastic Boolean satisfiability: A logical formalism for NEXPTIME decision problems with uncertainty,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, 5 2021, pp. 3877–3885.
- [7] L. Henkin, “Some remarks on infinitely long formulas,” *Journal of Symbolic Logic*, vol. 30, no. 1, pp. 167–183, 1961.
- [8] C. Cheng and J.-H. R. Jiang, “Lifting (D)QBF preprocessing and solving techniques to (D)SSAT,” in *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2023, pp. 3906–3914.
- [9] H.-R. Wang, K.-H. Tu, J.-H. R. Jiang, and C. Scholl, “Quantifier elimination in stochastic Boolean satisfiability,” in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, vol. 236. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 23:1–23:17.
- [10] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey, “ $\#\exists$ SAT: Projected model counting,” in *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Cham: Springer International Publishing, 2015, pp. 121–137.
- [11] Y.-W. Fan and J.-H. R. Jiang, “SharpSSAT: A witness-generating stochastic Boolean satisfiability solver,” in *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2023, pp. 3949–3958.

Synthesiz3 This: an SMT-Based Approach for Synthesis with Uncomputable Symbols

Petra Hozzová* and Nikolaj Bjørner†

*Czech Technical University, Prague, Czech Republic

†Microsoft, Redmond, USA



Abstract—Program synthesis is the task of automatically constructing a program conforming to a given specification. In this paper we focus on synthesis of single-invocation recursion-free functions conforming to a specification given as a logical formula in the presence of uncomputable symbols (i.e., symbols used in the specification but not allowed in the resulting function). We approach the problem via SMT-solving methods: we present a quantifier elimination algorithm using model-based projections for both total and partial function synthesis, working with theories of uninterpreted functions and linear arithmetic and their combination. For this purpose we also extend model-based projection to produce witnesses for these theories. Further, we present procedures tailored for the case of uniquely determined solutions. We implemented a prototype of the algorithms using the SMT-solver Z3, demonstrating their practical efficiency compared to the state of the art.

I. INTRODUCTION

Program synthesis is the task of finding a function satisfying the given specification. In this paper we focus on synthesis of recursion-free functions and require the solution to be guaranteed to be correct. To this end we lean on automated reasoning methods and do not consider machine learning. As synthesis subsumes verification, it is an inherently hard problem. Yet, recent developments in this area showed that fragments of the problem are tractable [15], [41], [39], [21].

There are many flavors of synthesis specifications. We consider specifications expressed as logical formulas capturing relational constraints on symbolically represented inputs/outputs, recently addressed by an SMT-based approach [39] and deductive synthesis in saturation [21]. Such specifications can use additional syntactic constraints: in [39] (and in the SyGuS paradigm in general [2]), the resulting function must syntactically conform to the provided grammar, while [21] introduces so-called *uncomputable* symbols which are not allowed to occur in the solution. However, while [21] uses uncomputable symbols, it does not formally characterize them. That motivated this work: we explore the semantics of specifications with uncomputable symbols, and we present an algorithm solving such specifications modulo (combinations of) theories.

Uncomputable symbols are a powerful tool allowing to make the specification more expressive, e.g. by (i) requiring the solution to be maximal, (ii) assuming the necessary condition for the existence of a total solution, or (iii) forcing the solution to discover properties of functions or predicates

used in the specification, but not allowed in the output. While [21] introduces specialized inference rules to handle the uncomputable symbols, we integrate them directly into the specification, expressed as a logical formula, by using second-order quantification. The challenges we then solve are how to *project away* these symbols from the formulas, and how to *find witnesses* capturing the solutions. To solve these challenges, we take an SMT-based route: we present an algorithm based on *quantifier elimination games* [7], [13], using and extending *model-based projections* [27], [6].

Contributions: In this work we synthesize recursion-free functions for first-order specifications using uncomputables and theories. Our main contributions are:

- 1) An algorithm for function synthesis in the presence of *uncomputable symbols* (Section III). In contrast to prior work [21], we use quantifiers to handle the scope of uncomputables. We present tailored model-based projection procedures (Section IV) used for quantifier elimination.
- 2) Our approach works directly for *partial function synthesis* (see Section III-C). That is, for specifications that are not realizable on all inputs, our approach identifies necessary and sufficient *pre-conditions* when solutions exist.
- 3) We introduce directed inferences exploiting the case when specifications have *unique solutions* (Section V).
- 4) We implemented a prototype of our method using the SMT-solver Z3 and compared it to the state of the art (Section VI).

II. PRELIMINARIES

We assume multi-sorted first-order and second-order logics and use standard definitions of their syntax and semantics. When discussing validity, we consider all formulas implicitly universally quantified. We work with theories as defined in SMT-LIB [5], in particular with the theories of equality and uninterpreted functions (EUF), and of linear integer and real arithmetic (LIA and LRA).

By $E[t]$ we signify that the term or formula E contains the term t , and by $E[t/y]$ we denote the term or formula we obtain from E by replacing every occurrence of the variable y by the term t . By $\simeq$ we denote the equality predicate, by $:=$ assignment, by $\mathbf{t}$ the tuple of terms (or variables/symbols) $t_1, \dots, t_n$, and by T_Σ the set of terms over the set of symbols Σ . We recall known notions we use in the paper: negation

normal form (NNF) of a formula F , denoted $NNF(F)$, is any formula equivalent to F , consisting only of literals, conjunctions, and disjunctions; disjunction normal form (DNF) is an NNF which is a disjunction of conjunctions of literals. We use the abbreviations sat/unsat for satisfiable/unsatisfiable. We abuse the notation and by $\top, \perp$ we denote the logical constants for true and false, as well as their values, and in our algorithms we also denote the undefined value by $\perp$. We reduce predicates into functions by replacing each atom $p(t)$ by $f_p(t) \simeq \top$, where f_p is a fresh function symbol with the domain $\{\perp, \top\}$. We write $\bar{\ell}$ for the complement of the literal ℓ . When C is a conjunction of literals $\bigwedge_{\ell \in \mathcal{L}} \ell$, we interchangeably refer to it as a set of its literals $\mathcal{L}$. Given a set of literals $\mathcal{L}$ and a formula F , where $\mathcal{L} \wedge F$ is unsat, we write $Core(\mathcal{L}, F)$ for $C \subseteq \mathcal{L}$ such that $C \wedge F$ is unsat.

We use “program” and “function” in the context of objects we synthesize as synonyms. By ϵ we denote the empty list and we use $|$ to denote list construction. We write $C \rightarrow r$ for the pair of a condition C and a term r used as one case of a program, and we represent the (possibly partial) program in the synthesis process as a list $C_1 \rightarrow r_1 \mid \dots \mid C_n \rightarrow r_n$. We construct the whole program from the list using the case constructor,

case $C_1 : r_1; \dots$ case $C_{n-1} : r_{n-1};$ default : r_n ,

which evaluates to the value r_i for $1 \leq i < n$ if $C_i \wedge \bigwedge_{j=1}^{i-1} \neg C_j$ is true, and to the value r_n otherwise. Since a list of pairs of conditions and terms uniquely defines a program, we sometimes abuse the notation and use the list in place of its corresponding program. Finally, we use $_$ to denote *any* value in the function – i.e., the case when any value of the correct sort satisfies the specification, and we use $_$ when there exists at least one computable term of that sort.

III. SYNTHESIS OF TOTAL AND PARTIAL FUNCTIONS

A. Specifications and Realizers

We focus on the class of synthesis specifications given as closed second-order logical formulas of the form

$$\exists f. \forall c, u. \varphi[f(c), c, u], \quad (1)$$

where f is the recursion-free function we want to synthesize, c are all symbols the function can depend on, and u are the symbols f should be independent of. Following the terminology of [21], we call c *computable* and u *uncomputable* symbols. The computable symbols include all symbols which are allowed in the solution for f : f ’s inputs, as well as constants, functions, and predicates. We call the terms and formulas over c *computable terms/formulas*, the terms/formulas containing symbols from u *uncomputable terms/formulas*, and the uncomputable symbols themselves also *uncomputables*.

Note that (1) specifies a *single single-invocation function*: f always occurs in φ with the same arguments c . Thus, specification (1) can be equivalently expressed in “unskolemized” (yet still possibly second-order) form as

$$\forall c. \exists y. \forall u. \varphi[y, c, u], \quad (2)$$

where the task is to find a witness for the first-order variable y . We can also allow y to be a tuple of variables, effectively specifying multiple functions, all depending on the same computable symbols/inputs.

In the rest of the paper, we use specifications like (2), formalized as follows:

Definition 1 (Total Function Synthesis). *Let Σ be a signature, let $u \subseteq \Sigma$ be the set of uncomputable symbols, let $y \in \Sigma \setminus u$, and let $c = \Sigma \setminus (\{y\} \cup u)$. A synthesis problem is a tuple*

$$\langle \Phi[y], u, y \rangle,$$

where $\Phi[y] = \varphi[y, c, u]$ is a quantifier-free formula called a synthesis specification, containing the variable y , which represents the function to be synthesized. A solution for the problem is a term $R \in T_c$, such that $\Phi[R/y]$ holds, called a realizer for y . As R defines a total function, we call this problem total function synthesis.

We consider theory-defined functions computable. In the following we always use y to represent the function to be synthesized, and sometimes write Φ for $\Phi[y]$, and when t does not contain y , we write $\Phi[t]$ for $\Phi[t/y]$.

Example 1 (2-knapsack). We encode the well-known problem: Given 2 items of weights w_1, w_2 and a capacity c , which of the items should we choose so that the total weight of the chosen items is maximal but at most c ? First, we define a formula specifying that y_1, y_2 form a valid pick from the two items, i.e., that the value of y_i is either 0 or w_i (for $i = 1, 2$), and that $y_1 + y_2$ is at most c :

$$\begin{aligned} \varphi[y_1, y_2] := & (y_1 \simeq 0 \vee y_1 \simeq w_1) \wedge (y_2 \simeq 0 \vee y_2 \simeq w_2) \\ & \wedge y_1 + y_2 \leq c \end{aligned}$$

Then the following synthesis problem specifies, assuming that c, w_1, w_2 are non-negative, that y is a valid pick from the two items, such that all valid picks u have lesser or equal combined weight than y :

$$\begin{aligned} & \langle (c \geq 0 \wedge w_1 \geq 0 \wedge w_2 \geq 0) \\ & \implies (\varphi[y] \wedge (\varphi[u] \implies u_1 + u_2 \leq y_1 + y_2)), \{u\}, y \rangle \end{aligned} \quad (3)$$

A realizer for (y_1, y_2) of (3) is:

case $w_1 + w_2 \leq c : (w_1, w_2);$
 case $w_1 \leq c \wedge (w_1 \geq w_2 \vee w_2 > c) : (w_1, 0);$
 case $w_2 \leq c \wedge (w_2 > w_1 \vee w_1 > c) : (0, w_2);$
 default : $(0, 0)$

B. The Expressivity of the Uncomputable Symbols

We define the scope of uncomputable symbols directly in the specification, by using explicit (and possibly second-order) quantification. This is in contrast with [21], which introduced uncomputable symbols, but forbade their use in the sought solution as an additional syntactic constraint, thus allowing the specifications to stay within first-order logic. These two methods of restricting the uncomputable symbols are, however, equivalent in what they allow to express.

The use of uncomputable symbols allows for more expressive specifications, which in turn leads to more interesting synthesis tasks. As in Example 1, with uncomputables we can require that the solution is optimal by changing the specification $\varphi[y, c]$ into $\varphi[y, c] \wedge (\varphi[u, c] \implies \text{is_better}(y, u))$, using a fresh uncomputable symbol u and some suitable definition of the predicate is_better . We can also use uncomputable symbols to ensure that a specification $\varphi[y, c]$ has a solution by adding an assumption $\varphi[u, c]$, allowing to turn partial function specifications into total ones. For example, there is no total solution for the specification $\langle 2y \simeq x, \emptyset, y \rangle$ over integers, but there is one for $\langle 2u \simeq x \implies 2y \simeq x, \{u\}, y \rangle$.¹

Further, u can include also function and predicate symbols. With those, synthesis can require discovering properties of uninterpreted functions or predicates:

Example 2 (Workshop [37], [19]). Consider the synthesis problem $\langle \Phi, \{w\}, y \rangle$, where:

$$\begin{aligned} \Phi := & ((\text{Mon} \vee \text{Sun}) \wedge (\text{Mon} \implies w(V)) \wedge (\text{Sun} \implies w(A))) \\ & \implies w(y) \end{aligned}$$

The example models a situation at a conference: given that it is Monday or Sunday today, and given that the workshop A is taking place on Sunday and the workshop V on Monday, the task is to synthesize a function that tells us which workshop is taking place. Additionally, the solution is not allowed to use the workshop predicate w – if we were able to evaluate $w(\cdot)$ ourselves, we arguably would not need to use any tools to synthesize the solution.

C. Partial Function Synthesis

We further consider *partial function synthesis*: a variation on the synthesis problem where the specification is not required to be satisfiable on all evaluations of computable symbols. Thus, the sought function is not required to be total.

Definition 2 (Partial Function Synthesis). *Let $\Phi[y]$ be a specification, where $\forall c. \exists y. \forall u. \Phi[y]$ is not necessarily valid. Then $\langle \Phi[y], u, y \rangle$ is a partial (function) synthesis problem. A solution for the problem is a pair $\langle C, R \rangle$, where C is a quantifier-free formula and R is a realizer for y , both over c , such that:*

$$\begin{aligned} C & \implies \Phi[R] & (4) \\ (\exists y. \forall u. \Phi) & \implies C & (5) \end{aligned}$$

The first condition ensures that $\Phi[R]$ holds under C , and the second condition ensures that $\langle C, R \rangle$ is a *maximal solution for partial function synthesis*.

While solutions $\langle C, R \rangle$ are recursively enumerable for first-order specifications, we note that they are in the undecidable $\forall\exists\forall$ -fragment [24]. The process of finding C that satisfies condition (5) relates to synthesizing uniform quantifier-free interpolants [42].

¹This trick does not work if φ already uses some uncomputable symbols. However, our synthesis algorithm (Section III-D) works for partial functions, as defined in Section III-C.

Example 3. Let $\Phi := q(y) \vee a \simeq b \simeq y$ with $u = \{q\}, c = \{a, b\}$. Then the weakest quantifier-free formula that implies $\forall u. \Phi$ is $a \simeq b \simeq y$, producing the partial solution $\langle a \simeq b, a \rangle$. On the other hand, for $\Phi := p(u) \vee a \simeq b \simeq y$, with $u = \{u\}, c = \{p, a, b\}$, the strongest $C \in T_c$ implied by $\forall u. \Phi$ is $\top$, so while $\langle a \simeq b, a \rangle$ satisfies (4), it does not satisfy (5).

Finite quantifier-free uniform interpolants do not always exist for combinations of theories:

Example 4. Let $\Phi := a \simeq y \wedge \Psi$, where Ψ does not have a uniform quantifier-free interpolant. For example, let Ψ be $\neg(p(u) \wedge a < u < b)$ for integers u, a, b , where $u = \{u\}$. Then there is no finite maximal solution to the synthesis problem, only an infinite one: $\langle \bigvee_{k>1} (a+k \simeq b \wedge \bigwedge_{0<i<k} \neg p(i+a)), a \rangle$.

Thus, we cannot develop complete algorithms in general. Yet, for theories admitting uniform interpolation, or even quantifier elimination, computing the strongest C satisfying (5) is possible. Our synthesis algorithm, presented next, finds partial solutions to (4) and uses (5) as a criterion for termination.

D. A Synthesis Algorithm

We present an algorithm for partial function synthesis. It iteratively builds a list of pairs $R := C_1 \rightarrow r_1 \mid \dots \mid C_n \rightarrow r_n$, stopping when $\forall u. \Phi \wedge \bigwedge_{1 \leq i \leq n} \neg C_i$ becomes unsatisfiable (step 2) – i.e., when the conditions $C_1, \dots, C_n$ cover all possible cases. Then the solution is $\langle \bigvee_{1 \leq i \leq n} C_i, R \rangle$. To compute one pair $C_j \rightarrow r_j$, we first find a formula $\pi^\forall[y]$ not containing symbols from u , which implies $\forall u. \Phi \wedge \bigwedge_{1 \leq i \leq j-1} \neg C_i$ (step 3). This $\pi^\forall[y]$ corresponds to a condition under which the specification holds, yet none of the other conditions found so far apply. Next we look for a witness r and a formula $\pi^\exists$, both only containing symbols from c , such that $\pi^\exists$ implies $\pi^\forall[r]$ (step 4). If we find such r and $\pi^\exists$, we set $C_j := \pi^\exists, r_j := r$ and add $C_j \rightarrow r_j$ to the list of pairs R (step 6). Otherwise we find a formula $\pi^\exists$ which implies $\exists y. \pi^\forall[y]$, and refine the specification by adding $\neg \pi^\exists$ to it (step 5). Then we continue with the next iteration: if $\forall u. \Phi \wedge \bigwedge_{1 \leq i \leq j} \neg C_i$ is satisfiable, we look for $C_{j+1} \rightarrow r_{j+1}$. Computation of $\pi^\forall$ and $\pi^\exists$ (steps 3 and 4) is addressed in Section IV.

The algorithm can also be used for total function synthesis by checking that the returned condition C is equivalent to $\top$.

Correctness of Algorithm 1 is straightforward:

Proposition 3. *Assume Algorithm 1 terminates with $\langle C, R \rangle$. If it never produced $r = \perp$ in step 4, then $\langle C, R \rangle$ is a maximal solution for the given partial synthesis problem. Otherwise $\langle C, R \rangle$ is a program satisfying condition (4) but there is no solution to (5).*

Proof. Each step preserves (4) $C \implies \Phi[R]$. The termination condition ensures (5) $\exists y. \forall u. \Phi \implies C \vee C^\perp$, where $C^\perp$ is the disjunction of projections for $r = \perp$.

Remark 4. *Algorithm 1 is complete for theories that admit quantifier elimination for computing $\pi^\forall, \pi^\exists$ in steps 3 and 4.*

Example 5. Our algorithm solves the workshop problem from Example 2. In the first iteration, $\forall w. \Phi$ is sat. We find $\pi^\forall[y] = \neg \text{Mon} \wedge \neg \text{Sun}$ (see Example 9 for details). Then a corresponding $\langle \pi^\exists, r \rangle$ is $\langle \neg \text{Mon} \wedge \neg \text{Sun}, _ \rangle$, where $_$ is the

Algorithm 1 Partial Function Synthesis

Given a partial function synthesis task $\langle \Phi, u, y \rangle$, return a solution $\langle C, R \rangle$.

- 1) Initialize $C := \perp$ and $R := \epsilon$.
 - 2) If $\forall u. \Phi \wedge \neg C$ is unsat then return $\langle C, R \rangle$.
 - 3) Find a satisfiable $\pi^\forall[y]$ over $c \cup \{y\}$, such that $\pi^\forall[y] \implies \forall u. \Phi \wedge \neg C$. (See Algorithm 3 in Section IV-B.)
 - 4) Find $\langle \pi^\exists, r \rangle$ such that $\pi^\exists$ is satisfiable and over c and either $\pi^\exists \implies \pi^\forall[r]$ with $r \in T_c$, or $r = \perp$ and there is no realizer over c for $\pi^\forall[y]$ and $\pi^\exists \implies \exists y. \pi^\forall[y]$. (See Algorithm 2 in Section IV-A.)
 - 5) If $r = \perp$, update $\Phi := \Phi \wedge \neg \pi^\exists$ and go to step 2.
 - 6) Otherwise, update $C := C \vee \pi^\exists$, $R := R \mid (\pi^\exists \rightarrow r)$ and go to step 2.
-

“any” realizer. We add $\neg Mon \wedge \neg Sun$ to C . As $\forall w. \Phi \wedge \neg C$ is satisfiable, we continue. We find $\pi^\forall[y] = A \simeq y \wedge Sun$, and then a corresponding $\langle \pi^\exists, r \rangle = \langle Sun, A \rangle$ (see Example 7). We add $\neg Sun$ to C . The formula $\forall w. \Phi \wedge \neg C$ is still satisfiable, so we continue. In the next iteration, $\pi^\forall = V \simeq y \wedge Mon$ and $\langle \pi^\exists, r \rangle = \langle Mon, V \rangle$. Then finally $\forall w. \Phi \wedge \neg C$ is unsat, and in fact, $C \equiv \top$. We thus return

$$\langle \top, (\neg Mon \wedge \neg Sun \rightarrow _) \mid (Sun \rightarrow A) \mid (Mon \rightarrow V) \rangle.$$

Since the first case in the realizer list uses $_$, we can choose to return V in that case. Further, since its condition $\neg Mon \wedge \neg Sun$ is the complement of the disjunction of the other two conditions (Sun or Mon), we can move it to the end of the list and then unite into a default case with the condition Mon , which also returns V , obtaining the final program:

$$\langle \top, \text{case } Sun : A; \text{default} : V \rangle$$

E. Unique Realizers

A useful special case in step 4 of Algorithm 1 is when a solution to the variable y is uniquely determined. For example, for a simplified 1-item version of knapsack, given by $c < 0 \vee w_1 < 0 \vee ((y_1 = 0 \vee y_1 = w_1) \wedge y_1 \leq c \wedge ((u_1 \neq 0 \wedge u_1 \neq w_1) \vee u_1 > c \vee u \leq y))$, and under the case $c \geq 0, w_1 \geq 0, u_1 = w_1, u_1 > c$, there is a unique solution to y given by 0. In Section V, we show how to read out unique solutions from a combination of the E-graph (a data structure capturing congruences, see Definitions 6 and 7) and arithmetical constraints. Our approach can be considered a generalization of E-graph saturation, where we consider also the theory of linear real arithmetic.

IV. QUANTIFIER PROJECTION

Our synthesis algorithm (Algorithm 1) uses two flavors of quantifier projection methods, one for existential and one for universal quantifiers. Prior work [6] explores projection operators for EUFLIA, but side-steps some requirements we have for the resulting formulas and realizers. We therefore introduce heuristics suitable for our setting. We describe

methods for existential projection in Section IV-A, and Section IV-B presents an approach for universal projection. We then plug these methods into Algorithm 1. For step 3 of Algorithm 1, we use universal projection (Algorithm 3), which takes $\Phi \wedge \neg C$ and returns a set of literals $\mathcal{L}$, which we assign to $\pi^\forall$. For step 4 of Algorithm 1, we use existential projection with witness extraction for the appropriate theory (for EUF, see Algorithm 2), which takes $\pi^\forall$ and an arbitrary model of it, and returns a suitable pair $\langle \pi^\exists, r \rangle$.

A. Existential Projection

To compute realizers, we use existential model-based projection.

Definition 5 ($\exists$ MBP and $\exists$ MBPR [27]). *Let Φ be a quantifier-free formula and assume $\mathcal{M} \models \Phi$. Then existential model-based projection $\exists$ MBP($\mathcal{M}, x, \Phi$) is a quantifier-free formula $\pi^\exists$ over symbols not in x , such that $\mathcal{M} \models \pi^\exists$ and $\pi^\exists \implies \exists x. \Phi$.*

Further, when y is a symbol in Φ , a model-based projection with realizer $\exists$ MBPR($\mathcal{M}, y, \Phi$) is a pair $\langle \pi^\exists, r \rangle$ such that $\pi^\exists$ is quantifier-free and satisfiable, $\pi^\exists \implies \Phi[r/y]$, and $y \notin \pi^\exists, r$.

Various algorithms for model-based projection have been integrated into Z3. We will use $\exists$ MBP to project uncomputable symbols in Section IV-B. We also introduce an algorithm for computing $\exists$ MBPR($\mathcal{M}, y, \mathcal{L}$) for EUF given a set of literals $\mathcal{L}$. Prior works [6], [26] develop projection algorithms that apply to combinations with arrays [25], integer arithmetic, and algebraic data-types [7]. However, they do not cover extracting computable realizers from projections. For example, [6] defines projections of first-order functions that can be defined without getting into notions of computable realizers. On the other hand, we can rely on witness extraction algorithms for LIA [31] for a combination with EUF.

Definition 6 (Congruence Preserving Partition). *Let $\mathcal{T}$ be a set of terms closed under sub-terms. A congruence preserving partition of $\mathcal{T}$ is a set $\mathcal{P}$ that partitions $\mathcal{T}$, such that for every $f(s), f(t) \in \mathcal{T}$, whenever $\forall i \in \{1, \dots, n\}. \exists P \in \mathcal{P}. s_i, t_i \in P$, then $\exists P' \in \mathcal{P}. f(s), f(t) \in P'$.*

Definition 7 (E-graph, E). *An E-graph, E, is a data structure that provides access to a congruence preserving partition. It maps each term $t \in \mathcal{T}$ to its partition by $\text{root}(t)$.*

Every interpretation $\mathcal{M}$ induces a congruence preserving partition. Conversely, for a set of equalities over EUF, there is a unique congruence preserving partition $\mathcal{P}$ such that for each equality $s \simeq t$, s, t belong to the same class in $\mathcal{P}$, and every other partition with that property is obtained by taking unions of classes in $\mathcal{P}$. When a solver constructs an E-graph E, then every equality that is added to E is justified by a set of literals. Justification extends to every implied equality and can be represented as a proof tree that uses axioms for EUF. Thus, when $\text{root}(t) = \text{root}(t')$, then the function $\mathcal{J}(t, t')$ returns the set of equality literals used to establish the equality between t, t' in E. We use the partition to find a computable term $t' \in T_c$ equal to a given term t . To this end, we define a

so-called representative function rep_c , returning a suitable t' for the given t , as well as for the class t belongs to in $\mathcal{P}$.

Definition 8 ($\text{rep}_c(P)$ [6], [14]). *Given a congruence preserving partition $\mathcal{P}$, a representative, overloaded as: $\text{rep}_c : 2^{\mathcal{T}} \rightarrow T_c \cup \{\perp\}$, $\text{rep}_c : \mathcal{T} \rightarrow T_c \cup \{\perp\}$, is a function satisfying:*

- 1) $\text{rep}_c(t) = \text{rep}_c(P)$ for each $P \in \mathcal{P}, t \in P$.
- 2) $\text{rep}_c(P) = \perp$ or $\text{rep}_c(P) = f(\text{rep}_c(t_1), \dots, \text{rep}_c(t_n)) \in T_c$ such that $f(t) \in P, \text{rep}_c(t_i) \neq \perp$ for each t_i .
- 3) *Maximality: for every $P \in \mathcal{P}$, $\text{rep}_c(P) = \perp$ implies that every term $f(t)$ in P either has $f \notin c$, or $\text{rep}_c(t_i) = \perp$ for some i .*

Searching for a realizer of y for $\mathcal{L}[y]$ can be performed by Algorithm 2.²

Algorithm 2 $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ for EUF

Given $\mathcal{L}[y]$ over $c \cup \{y\}$, and $\mathcal{M}$ such that $\mathcal{M} \models \mathcal{L}[y]$, return $\langle \pi^\exists, r \rangle$ over c such that either $\pi^\exists \implies \mathcal{L}[r]$, or $r = \perp$ and $\pi^\exists \implies \exists y. \mathcal{L}[y]$.

- 1) Let $\mathcal{P}$ be a congruence preserving partition, based on $\mathcal{M}$, satisfying $\mathcal{L}$.
 - 2) Let $\mathcal{C} := \{\text{rep}_c(P) \mid P \in \mathcal{P}, \text{rep}_c(P) \neq \perp\} \cup \{f(t) \mid f \in c, t_i = \text{rep}_c(P_i) \neq \perp\}$.
 - 3) If there is $r \in \mathcal{C}$ such that $\mathcal{L}[y] \wedge y \simeq r$ is satisfiable, return $\langle \mathcal{L}[r], r \rangle$.
 - 4) Otherwise, there is no computable r such that $\mathcal{L}[r]$ does not contradict equalities following from $\mathcal{M}$. Return $\langle \exists\text{MBP}(\mathcal{M}, y, \mathcal{L}), \perp \rangle$.
-

Proposition 9 (Correctness and Completeness of Algorithm 2). *Algorithm 2 produces $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ iff one exists, and returns $\langle \exists\text{MBP}(\mathcal{M}, y, \mathcal{L}), \perp \rangle$ otherwise.*

Proof. The goal is to establish if there exists a term $r \in T_c$ such that $\mathcal{L}[r] \equiv \mathcal{L}[y] \wedge y \simeq r$ is satisfiable. For this purpose we enumerate a subset of T_c that is necessary and sufficient. In particular, it is sufficient to check the set of terms in $\mathcal{P}$ that are already represented by computable terms, and terms immediately obtained from these terms by using the signature c . Suppose there is a term $r \in T_c$ such that $\mathcal{L}[r]$ is satisfiable. Then either it is congruent to a term in $\text{rep}_c(P), P \in \mathcal{P}$, or we can trim it down to a term with immediate children in $\text{rep}_c(P)$. Suppose $r = f(t)$ is a solution, but t_i for some i is not congruent within $\mathcal{P}$. Then we could use t_i in place of r because we can produce a model of $\mathcal{L}[t_i]$ by setting the interpretation of t_i the same as $f(t)$. Note that it is possible that $\mathcal{M}$ enforces congruences not in $\mathcal{P}$ so it is not necessarily the case that $\mathcal{M} \models \mathcal{L}[r]$.

Remark 10. *Single invocations of $\exists\text{MBPR}$ do not necessarily produce formulas that are equivalent to $\exists y. \mathcal{L}[y]$. Thus, $\mathcal{L}[y] \wedge \neg \mathcal{L}[r]$ is not necessarily unsatisfiable. However, $\exists\text{MBPR}$ for EUF+LIA finitely eliminates all models of $\mathcal{L}$.*

Example 6. In the following constraints, there are no solutions for y using T_c . (1) Let $c = \{a\}$ and $\mathcal{L} = y \not\approx a$. (2) Let $c = \{f, a\}$ and $\mathcal{L} = f(a) \simeq a \wedge y \not\approx a$.

²We describe the practical optimizations for Algorithm 2, including finding the “any” realizer $\perp$, in Section VI.

Example 7. We compute $\langle \pi^\exists, r \rangle$ for $\mathcal{L}[y] = \{A \simeq y, \text{Sun}\}$, requested in Example 5. A partition $\mathcal{P}$ based on $\mathcal{L}$ is $\{\{A, y\}, \{\text{Sun}, \top\}\}$. Then $\mathcal{C} = \{A, \text{Sun}\}$. Since A is of the suitable sort and $\mathcal{L}[A] \wedge y \simeq A$ is satisfiable, the result is $\langle \text{Sun} \wedge A \simeq A, A \rangle$, equivalent to $\langle \text{Sun}, A \rangle$.

Extending MBP from LIA to EUFLIA requires enforcing Ackermann axioms. We derive a recipe from [6].

Definition 11 ($\exists\text{MBPR}$ for LIA + EUF). *Let $\mathcal{M} \models \mathcal{L}$ over LIA + EUF, and assume $\mathcal{L}$ is closed under Ackermann reductions with respect to y . That is, for every occurrence $f(s)$ and $f(t)$, where f is uninterpreted, s, t are arithmetic terms that contain y : either $(s \not\approx t) \in \mathcal{L}$ or $(f(s) \simeq f(t)) \in \mathcal{L}$. Then $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ is defined by LIA projection where only occurrences of y within arithmetic terms are considered.*

Remark 12. *We established complete $\exists\text{MBP}$ and $\exists\text{MBPR}$ methods for EUF and EUFLIA. $\exists\text{MBP}$ for combinations of theories that include arrays and algebraic datatypes are implemented in Z3.*

B. Universal Projection

Universal projection for EUFLRA was addressed in different guises. Covering algorithms by [16] can be used to compute projections for EUFLRA. They provide a strongest Ψ such that $(\exists u. \neg \Phi) \implies \Psi$. Equivalently, $\neg \Psi \implies \forall u. \Phi$. We would like to unfold $\neg \Phi$ incrementally as a DNF so that we can compute implicants of $\forall u. \Phi$ incrementally and terminate once we find a suitable projection.

We here outline a method that produces a DNF of implicants. It can be used for ground formulas and combinations of theories where $\exists\text{MBP}$ applies. It derives from the MARCO algorithm [32] for core enumeration, where our twist is to apply projection to cores during enumeration and to terminate early when a suitable satisfying assignment is found. Let Φ be a ground formula and $\mathcal{L}$ be all the literals that occur in the negation normal form of $\neg \Phi$. The set of cores over $\mathcal{L}$ with respect to Φ can be enumerated using core enumeration techniques. Each core $\mathcal{C}$ is such that $\mathcal{C} \wedge \Phi$ is unsatisfiable, which means $\Phi \implies \neg \mathcal{C}$. We can filter out cores that are unsatisfiable by themselves. It remains to eliminate u from each core, or in other words, find $\mathcal{C}'$, such that $(\forall u. \neg \mathcal{C}) \implies \neg \mathcal{C}'$. This question is equivalent to finding $\mathcal{C}'$, such that $\mathcal{C}' \implies \exists u. \mathcal{C}$. We can use $\exists\text{MBP}$ to establish $\mathcal{C}'$ based on a model for $\mathcal{C}$. Thus,

$$\begin{aligned} \forall u. \Phi &\equiv \forall u. (\Phi \wedge \bigwedge_i \neg \mathcal{C}_i) \\ &\equiv (\forall u. \Phi) \wedge \bigwedge_i \forall u. \neg \mathcal{C}_i \\ &\equiv (\forall u. \Phi) \wedge \bigwedge_i \neg \mathcal{C}'_i \end{aligned}$$

We have found a sufficient set of cores when (if) the formula

$$\left(\bigwedge_i \neg \mathcal{C}'_i \right) \wedge \neg \forall u. \Phi$$

is unsat. But we do not need to compute the full set of cores before we can extract candidates for $\pi^\forall$. We can return a propositional model $\pi^\forall$ of the current partial set of projections $\bigwedge_i \neg \mathcal{C}'_i$, whenever $\pi^\forall \wedge \neg \forall u. \Phi$ is unsat. In other words, it

suffices to extract a DNF of $\bigwedge_i \neg C'_i$ on demand without having the full $\bigwedge_i \neg C'_i \implies \forall \mathbf{u}. \Phi$.

This discussion motivates Algorithm 3. It enumerates propositional models of literals in Φ , using additional constraints $\mathcal{O}$ (blocking implicants of $\neg\Phi$) and $\mathcal{S}$ (blocking implicants of Φ which use uncomputable symbols), both initialized to $\top$. The models are constructed by first finding a subset U of uncomputable literals in $\neg\Phi$ consistent with $\mathcal{O} \wedge \mathcal{S}$, and then finding a model for $U \wedge \Phi$. If this model satisfies a set of literals $\mathcal{L}_c^\top$ not using uncomputables, and $\mathcal{L}_c^\top \implies \Phi$, we are done (step 7). If $U \wedge \Phi$ is unsat, we block this model in future enumeration: we project $\mathbf{u}$ from the unsat core $\text{Core}(U, \Phi)$, negate the projection obtaining $\bigvee_{\ell \in \bar{\mathcal{C}}} \ell$ (corresponding to $\neg C'_i$ from the discussion above), add this clause to $\mathcal{O}$, and add the new literals from $\bar{\mathcal{C}}$ to all clauses in $\mathcal{S}$ (step 4). The last case is when the propositional model implies Φ , but using an implicant which includes uncomputable literals. To block this model in future enumeration iterations, we add a clause over literals that are false in this model to $\mathcal{S}$ (step 8). In contrast to MARCO, the set of literals that is used to enumerate cores grows (in step 4). To allow models using the new literals, the algorithm weakens $\mathcal{S}$ by including new literals from $\bar{\mathcal{C}}$. The algorithm is resumable after returning in step 7 by supplying a clause $\bigvee_{\ell \in \bar{\mathcal{C}}} \ell$ implied by $\neg\pi^\forall$ to step 4(a).

Algorithm 3 Universal projection using cores, correction sets and $\exists\text{MBP}$

Given Φ and $\mathbf{u}$, return $\pi^\forall$ such that either $\pi^\forall \implies \forall \mathbf{u}. \Phi$, or $\pi^\forall = \perp$.

- 1) Let $\mathcal{O} := \top, \mathcal{S} := \top$, let $\mathcal{L}_u$ be literals in $\text{NNF}(\neg\Phi)$ that contain symbols from $\mathbf{u}$, and let $\mathcal{L}_c$ be the literals in $\text{NNF}(\Phi)$ that do not contain symbols from $\mathbf{u}$.
 - 2) If $\mathcal{O} \wedge \mathcal{S}$ is unsat, then return $\perp$.
 - 3) Find a subset $U \subseteq \mathcal{L}_u \cup \mathcal{L}_c$ that is satisfiable with model $\mathcal{M}$ and implies $\mathcal{O} \wedge \mathcal{S}$.
 - 4) If $U \wedge \Phi$ is unsat, let $\mathcal{C} := \text{Core}(U, \Phi)$, let $\bar{\mathcal{C}} := \{\bar{\ell} \mid \ell \in \exists\text{MBP}(\mathcal{M}, \mathbf{u}, \mathcal{C})\}$, and
 - a) add $\bigvee_{\ell \in \bar{\mathcal{C}}} \ell$ to $\mathcal{O}$, update $\mathcal{S} := \{S \vee \bigvee_{\ell \in (\bar{\mathcal{C}} \setminus \mathcal{L}_c)} \ell \mid S \in \mathcal{S}\}$, $\mathcal{L}_c := \mathcal{L}_c \cup \bar{\mathcal{C}}$, and
 - b) go to step 2.
 - 5) Otherwise, let $\mathcal{M}'$ be such that $\mathcal{M}' \models U \wedge \Phi$.
 - 6) Let $\mathcal{L}_c^\top := \{\ell \in \mathcal{L}_c \mid \mathcal{M}'(\ell) = \top\}$
 - 7) If $\mathcal{L}_c^\top \wedge \neg\Phi$ is unsat, then $\mathcal{L}_c^\top$ is an implicant of Φ . Return $\pi^\forall := \text{Core}(\mathcal{L}_c^\top, \neg\Phi)$.
 - 8) Let $\mathcal{L}^\perp := \{\ell \in \mathcal{L}_c \cup \mathcal{L}_u \mid \mathcal{M}'(\ell) = \perp\}$, add $\bigvee_{\ell \in \mathcal{L}^\perp} \ell$ to $\mathcal{S}$, and go to step 2.
-

Proposition 13 (Correctness of Algorithm 3). *If Algorithm 3 returns $\pi^\forall \neq \perp$, then $\pi^\forall \implies \forall \mathbf{u}. \Phi$.*

Proof. Since the algorithm can only return $\pi^\forall$ from step 7, it means $\pi^\forall = \text{Core}(\mathcal{L}_c^\top, \neg\Phi)$ where $\mathcal{L}_c^\top$ does not contain symbols from $\mathbf{u}$, and $\pi^\forall \implies \Phi$, which then entails $\pi^\forall \implies \forall \mathbf{u}. \Phi$.

Remark 14. *The algorithm provides a way to incrementally compute cases of a partial function. It is complete for theories that admit quantifier elimination.*

Proof. Claim: the sets $\mathcal{L}_c$ obtained from the NNF of Φ and $\mathcal{L}_u$ obtained from the NNF of $\neg\Phi$ are sufficient for enumerating conflicts with Φ , and $\neg\Phi$, respectively. If L is a propositional model such that $L \wedge \Phi$ is unsat then $L' := L \setminus \{\ell \in L \mid \ell \notin \mathcal{L}_u, \bar{\ell} \in \mathcal{L}_u\}$ also satisfies $L' \wedge \Phi$ is unsat. The symmetric claim is: If L is a propositional model such that $L \wedge \neg\Phi$ is unsat, then $L' := L \setminus \{\ell \in L \mid \ell \notin \mathcal{L}_c, \bar{\ell} \in \mathcal{L}_c\}$ also satisfies $L' \wedge \neg\Phi$ is unsat. Justification (for the last claim): If $\bar{\ell}$ occurs positive in Φ , but ℓ does not occur in Φ at all, then ℓ does not occur negatively in $\neg\Phi$ and therefore doesn't contribute to $L \wedge \neg\Phi$ being unsat.

Claim: All propositional models of $\bigwedge_i \neg C'_i$ are covered by step 7. Here, C'_i is a projection of C_i and C_i is a propositional model of Φ . It follows as step 4 prunes literal combinations that are propositional models of $\neg\Phi$, while step 7 identifies the models for $\bigwedge_i \neg C'_i$, and step 8 forces enumeration of propositional models to contain literals that belong to a correction set of non-models of Φ .

Put together, if the underlying theory allows to finitely eliminate the propositional models (in step 4 by quantifier elimination), we eventually reach the model corresponding to the implicant of $\forall \mathbf{u}. \Phi$.

Example 8. Let $\Phi := f(u) \simeq a \implies f(u) \simeq y$ where $\mathbf{c} = \{f, a\}$, $\mathbf{u} = \{u\}$. Let $U := \{f(u) \simeq a, f(u) \not\simeq y\}$. Then $U \wedge \Phi$ is unsat and the core $\mathcal{C} := f(u) \simeq a \wedge f(u) \not\simeq y$ with respect to Φ has the projection $a \not\simeq y$. We add $a \simeq y$ to $\mathcal{O}$, and now $\mathcal{O} \wedge \neg\Phi$ is unsatisfiable and we can return $a \simeq y$ in step 7. Existential projection computes the pair $\langle \top, a \rangle$, which is a solution to the synthesis problem.

Example 9. We compute $\pi^\forall$ requested by Algorithm 1 in Example 5. We have $\Phi := ((\text{Mon} \vee \text{Sun}) \wedge (\text{Mon} \implies w(V)) \wedge (\text{Sun} \implies w(A))) \implies w(y)$ and $\mathbf{u} = \{w\}$. First let $U := \{\neg\text{Mon}, \text{Sun}, w(A), w(V), \neg w(y)\}$. Then $U \wedge \Phi$ is unsat, with $\mathcal{C} = \text{Core}(U, \Phi) = \{\text{Sun}, w(A), \neg w(y)\}$, projected to $\{\text{Sun}, A \not\simeq y\}$. We let $\mathcal{O} := \neg\text{Sun} \vee A \simeq y$ and find a new $U := \{\neg\text{Mon}, \neg\text{Sun}, w(A), w(V), \neg w(y)\}$. Then $U \wedge \Phi$ is sat, and we find its model $\mathcal{M}' = \{\text{Sun} \mapsto \perp, \text{Mon} \mapsto \perp, A \mapsto v_0, V \mapsto v_1, y \mapsto v_2, w \mapsto \lambda v.v \neq v_2\}$. We get $\mathcal{L}_c^\top = \{\neg\text{Mon}, \neg\text{Sun}\}$, which implies Φ , and we thus return $\pi^\forall = \text{Core}(\mathcal{L}_c^\top, \neg\Phi) = \{\neg\text{Mon}, \neg\text{Sun}\}$.

V. SYNTHESIZING UNIQUE FUNCTIONS

When specifications have unique realizers, we can take important shortcuts. For synthesis tasks based on Definition 2, there may be unique solutions to y based on implicants $\pi^\forall$ of Φ . In that case, we can bypass invoking $\exists\text{MBPR}$. It is well-recognized in QBF solving [36] that there can be a substantial advantage to detecting when a formula entails that there is a unique solution for an existential variable. Heuristics based on unique solutions are also used to significantly aid synthesis of Boolean functions [1]. We here outline how we leverage unique function specifications in the case of EUF and LRA.

The setting we will be discussing is the following: given a variable y , we introduce inferences that can establish when a set of constraints Φ entails that y is equal to a term t that does not contain any of the symbols from $\{y\} \cup \mathbf{u}$. For LRA in isolation, there is a canonical method for inferring equalities from conjunctions of equalities and inequalities: Gaussian elimination. For EUF in isolation, we can consult rep_c from Definition 8 if the congruence closure induced by a conjunction of equalities implies that y is congruent to a term without symbols from $\{y\} \cup \mathbf{u}$. We summarize these methods in a bit more detail next. Our main observation is that the *combination* of EUF and LRA is also amenable to synthesizing unique solutions. In Section V-C we show how to extract unique solutions for y (if they exist) in a combination of the two theories. The main claim is that Gaussian elimination can be interleaved with congruence closure rules to establish all equalities modulo LRA + EUF under a set of asserted literals. It builds on an idea of using Gaussian elimination to solve for symbols that should be eliminated and use the resulting equations to augment the congruence closure for equalities that may have not been surfaced by the LRA solver. Recall, that it is enough for theory combinations to agree on equalities with respect to a candidate model [12] and not all implied equalities. So our method extracts implied equalities on the fly from LRA and takes them into account by augmenting the congruence closure. Beyond experimenting with this method on our benchmarks, this approach also lets Z3 expose equality saturation modulo EUF and LRA.

A. EUF

The heart of CDCL(T) solvers is a congruence graph E for the set of terms in Φ . In a satisfiable state, E maintains a set of currently implied congruences. If in a satisfiable state $\text{rep}_c(y) \in T_c$ and $\mathcal{J}(y, \text{rep}_c(y))$ has no terms from $\mathbf{u}$, then we have a justification that y is equal to a computable term using assumptions that are computable or use y . Assuming the state is consistent for EUF, the justification $\mathcal{J}(y, \text{rep}_c(y))$ is also EUF-consistent. We then produce the condition $\mathcal{J}(y, \text{rep}_c(y))[\text{rep}_c(y)/y]$ and the realizer $\text{rep}_c(y)$. By construction, this approach extracts realizers that are forced by E , but it may fail to detect that a set of constraints have only one possible realizer:

Example 10. Let $\Sigma = \{a, b\}$, $\mathcal{L} := y \neq b \wedge a \neq b$. Then a is a unique realizer for y , but for the congruence class over $\mathcal{L}$, $\text{rep}_c(y) = \perp$.

B. LRA

It is also possible to solve for variables over LRA. We recall the dual tableau normal form used by the arithmetic solver. A tableau T maintains a set of basic, $\mathcal{B}$, and non-basic, $\mathcal{N}$, variables such that each basic variable is in a solved form from the non-basic variables. The sets $\mathcal{B}$ and $\mathcal{N}$ are disjoint.

$$T : \bigwedge_{b \in \mathcal{B}} \left(x_b = \sum_{i \in \mathcal{N}} a_{bi} x_i \right)$$

Furthermore, it maintains bounds B on all variables $\bigwedge_i \text{lo}(x_i) \leq x_i \leq \text{hi}(x_i)$, where lower $\text{lo}(x_i) \in \{-\infty\} \cup \mathbb{R}$,

$\text{hi}(x_i) \in \{\infty\} \cup \mathbb{R}$. Every lower and upper bound that is not infinite is justified by a set of literals $\mathcal{J}_{\text{lo}}(x)$, $\mathcal{J}_{\text{hi}}(x)$. A feasible tableau comes with an evaluation function $\text{val}(x_i)$ that maps every variable into a value from $\mathbb{R}$, such that:

$$\text{val}(x_b) = \sum_{i \in \mathcal{N}} a_{bi} \text{val}(x_i) \quad \text{for every } b \in \mathcal{B}$$

Let x be an arithmetic variable. It can be reduced to a solution in terms of other variables or constants in the following ways:

- 1) $x = \text{lo}(x)$, when x is fixed by existing bounds $\text{lo}(x) = \text{hi}(x)$.
- 2) $x = \text{val}(x)$, when the current value for x cannot be changed, i.e., when $(x > \text{val}(x) \vee x < \text{val}(x)) \wedge T \wedge B$ is infeasible.
- 3) $x = \sum_i a_i x_i$, when $x \in T$ and $T', x = \sum_i a_i x_i$ is obtained from T by pivoting x to a base variable.

C. EUF + LRA

Suppose there is a term $t \in T_c$ such that $\mathcal{L}$ entails $y \simeq t$. Based on the ingredients for EUF and LRA we establish a proof search for t . Represent $\mathcal{L}$ as $E + T + B$, then the search succeeds by deriving the sequent $E, T, B \vdash y \rightsquigarrow t$ with $t \in T_c$, using the following rules:

$$\begin{array}{c} \frac{E, T, B \vdash t_i \rightsquigarrow s_i \text{ for each } i}{E, T, B \vdash \sum_i a_i t_i \rightsquigarrow \sum_i a_i s_i} \\[10pt] \frac{\text{rep}_c(t) \neq \perp}{E, T, B \vdash t \rightsquigarrow \text{rep}_c(t)} \quad \frac{t \text{ is fixed by } T, B}{E, T, B \vdash t \rightsquigarrow \text{val}(t)} \\[10pt] \text{Congruence} \frac{E \vdash t = f(\mathbf{t}), f \in \mathbf{c} \quad E, T, B \vdash t_i \rightsquigarrow t'_i \text{ for each } i}{E, T, B \vdash t \rightsquigarrow f(\mathbf{t}')} \\[10pt] \text{Gauss} \frac{T \equiv t = \sum_i a_i t_i, T' \quad E, T', B \vdash t \rightsquigarrow t'}{E, T, B \vdash \sum_i a_i t_i \rightsquigarrow t'} \end{array}$$

Proposition 15. *For a real-valued variable y , the search for $t \in T_c$ with $E, T, B \vdash y \rightsquigarrow t$ saturates finitely and finds a solution in T_c if and only if it exists.*

Proof (Outline). The proof search is finite because every time it solves using T it removes the row for the solved variable through a Gauss elimination step, and the number of terms in E to check is finite. Assuming arithmetic operations are computable, the rule that decomposes summations is a special case of the congruence rule. The proof search is also complete for arithmetical LRA since if it is invoked with y and finitely saturates without producing a term for y , we can establish a model where y is distinct from every term in T_c : Set $\text{val}(y) := \text{val}(y) \pm \epsilon$ for an infinitesimal ϵ , and inductively, for extensions $t = \sum_i a_i t_i$ there is at least one term t_i that cannot be solved for, so set $\text{val}(t_i) := \text{val}(t_i) \pm \epsilon/a_i$. The sign $\pm \epsilon$ is chosen to make sure the new value of y is within the bounds for y in B . Whenever t_i is congruent to $f(\mathbf{t})$, $f \in \mathbf{c}$, there is inductively an argument t_i which is non-solvable, so the interpretation of f

can be adjusted to take into account the new value for t_i . This produces a non-standard model where the interpretation of all terms in T_c are over the rational numbers while y evaluates to a non-standard real. Functions are interpreted over rationals if all arguments are rationals, and can otherwise evaluate to a non-standard real if some argument is not a real number.

We note that it is straightforward to extend the inference system to also extract justifications by tracking inequalities used for fixed bounds and equalities used for congruences.

VI. IMPLEMENTATION AND EVALUATION

We implemented our method as a proof of concept, and compared it with state-of-the-art methods supporting uncomputables: the synthesis mode of Vampire [21] and the leading SyGuS solver cvc5 [39], obtaining encouraging early results.

A. Implementation

We implemented Algorithms 1, 2, and 3, as well as the unique solution finding of Section V, as a stand-alone Python prototype (available online at [20]) using the Z3 API, which we also extended by new features. We use variables of array sort for higher-order quantification. The theory of arrays in Z3 coincides with SMT-LIB 2 [5] for ground formulas, but not quantified formulas. E.g., the first-order theory of array select function admits models where select does not include bijections, but with Z3 one can instantiate quantified array variables by $\lambda x.x$ and select is just a function application.

The API already provided functions for computing $\exists\text{MBP}$, an incomplete method for $\exists\text{MBPR}$, and accessing the E-graph. We extended Z3 by exposing E-graph justifications, and by a new heuristic for finding witnesses from T_c for EUF in $\exists\text{MBPR}$. It searches for an E-match for y that does not lead to equating shared sub-terms, and specifically avoids merging sub-terms used in disequalities. We also added a method for solving variables that is complete for LRA and extends to LIA (incompletely, not to deal with non-convexity of LIA).

We optimize all our algorithms as follows. In Algorithm 1, step 6, if $u = \emptyset$, we use $\Phi[r/y]$ as the condition for realizer r instead of $\pi^\exists$. Further, to compute $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ of Algorithm 2, we first check if $\mathcal{L}$ contains y , and if it does not, we return the “any” realizer (as in Example 5). Otherwise, we try calling $\exists\text{MBPR}$ of the API, and if that does not return a realizer, we iterate over the candidates provided by rep_c , only producing a new one if the last did not work out. Finally, in Algorithm 3, step 7, we check the unsatisfiability with respect to $\neg\Phi'$, where Φ' is (i) initialized to the original specification Φ , and (ii) only if Algorithm 3 returns the same $\pi^\forall$ twice, updated to $\Phi' := \Phi \wedge \neg C$ using the current C of Algorithm 1.

B. Evaluation

To the best of our knowledge, there was no standard benchmark set for synthesis with uncomputable symbols as of writing this paper. However, a dataset based on the benchmarks from [21] was under development in parallel with this paper,

Id	Benchmark subset description	Count	Has u	Partial	Unique
I	simple if-then-else benchmarks from SyGuS competition [3]	3	✗	✗	✓
II	k -knapsack (see Example 1)	5	✓	✗	✓
III	maximum function for n given integers	10	✗	✗	✓
IV	lower bound function for n integers	10	✗	✗	✗
V	function finding an integer lower than all n given integers	10	✗	✗	✗
VI	function finding an integer between the given bounds, <u>assuming</u> some exists	4	✓	✗	✗
VII	function finding an integer between the given bounds, <u>if</u> some exists	4	✓	✓	✗
VIII	function finding a solution for y in the given equation, assuming it exists	3	✓	✗	✓
IX	function finding a solution for y in the given equation, <u>if</u> it exists	3	✗	✓	✓
X	Examples 2 (and a computable variant thereof) and 8 (and a variant thereof)	4	✗+✓	✗	✓
XI	the first specification or Example 3	1	✓	✓	✓

TABLE I: Summary of the 57 benchmarks used: set id, description, how many benchmarks are in the subset, and whether they use uncomputables, specify a partial function, and a unique function. Benchmark sets I-IX use LIA, X and XI use UF. In set II we have $k \in \{1, 2, 3, 4, 5\}$, and in sets III-V we have $n \in \{2, 5, 10, 15, 20, 25, 30, 50, 100, 150\}$.

	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	sum	total & partial
Synthesiz3	3	3	9	4	3	4	4	3	3	4	1	33	& 8
Vampire	3	0	5	2	0	4	–	0	–	2	–	16	& –
cvc5	3	1	5	5	5	4	–	0	–	4	–	27	& –

TABLE II: Experimental results: numbers of problems solved by each solver in benchmark sets I-XI. Benchmark sets VII, IX, and XI specify partial synthesis problems not supported by Vampire nor cvc5, which we denote by “–”.

and in the meantime it was accepted for publication [18].³ We took the subset of this dataset supported by our method,⁴ and extended it with the solvable examples from this paper, as well as additional partial function synthesis problems. Table I shows a summary of the benchmarks, organized into 11 subsets denoted I-XI. The set contains 57 benchmarks: 18 small, conceptually distinct problems (all 3 problems of subset I, 2 per both VI and VII, 4 in X, and one per all the other subsets), and 39 scaled-up versions of the small instances.

We evaluated the performance of our prototype on this dataset and for the total synthesis problems also compared

³The dataset was co-developed by one of the authors of this paper. It is available at https://github.com/vprover/vampire_benchmarks/tree/master/synthesis. The version that was available shortly before this paper was submitted, and which we used for evaluation, is at https://github.com/vprover/vampire_benchmarks/tree/6356e52/synthesis.

⁴We excluded problems using (i) theories beyond EUFLIRA, or (ii) more complex quantification, or (iii) interpreted uncomputables, or (iv) which were too similar to others in the dataset – e.g. we used maximum but not minimum.

it to Vampire and cvc5.⁵ We ran each benchmark on a single core of a 1.60GHz CPU with 16 GB RAM, with the time limit of 1 minute. We chose 1 minute as it is enough time to solve the solvable small problems, and show the trend for the scaled-up problem instances, but not solve all the scaled-up instances. Table II shows the numbers of solved problems. Successful solving of the small instances took all solvers less than 1 second. For the larger instances, the runtimes increased up to the time limit. We note that what makes many scaled-up instances hard is that they encode problems with many symmetries: e.g., each case of the maximum-of- n -integers function has the same structure, but with different variables.

Overall, our approach solves the most total function synthesis problems, and also partial synthesis problems which cvc5 and Vampire do not support. Our approach scales as well as or better than both Vampire and cvc5 for all benchmarks with a unique solution, and slightly worse than cvc5 on problems with infinitely many possible solutions (sets IV and V). Notably, within 1 minute, only our approach solves 2- and 3-knapsack, and it also finds the maximum-of- n -integers function for n up to 100, while cvc5 and Vampire can go only up to $n \leq 20$. Interestingly, both cvc5 and Vampire fail to solve all benchmarks of set VIII, including the smallest instance $\langle 2u \simeq x \implies 2y \simeq x, \{u\}, y \rangle$, “find the half of an even x ”.

VII. RELATED WORK

Synthesis: The field of synthesis encompasses very different approaches. We only overview methods similar to ours: synthesizing functions from scratch based on specifications given as logical formulas with some syntactic restrictions. The general restriction setting in [40] is to integrate a black-box function within a solver that determines whether a term is computable. We base our specification format on [21]. In contrast with [21], we distinguish uncomputable symbols directly in the specification formula by using quantification. The deductive synthesis methods of [21], [40] are based on theorem proving, and [21] is implemented in Vampire [29]. While these methods introduce specialized inference rules to filter terms using uncomputable symbols, our approach uses quantifier elimination games to remove uncomputables, and thus requires no changes in the underlying search procedures.

Another related paradigm is SyGuS [2], which supports specifying multiple functions with different inputs, and constraining the space of solutions to functions generated by a given grammar. SyGuS solvers include cvc5 [4], [39] and DryadSynth [22]. Compared to the specification setting we consider, SyGuS allows a more fine-grained syntactic control over the synthesized function. We note that to solve such specifications, we could generate candidate terms using the provided grammar instead of using $\exists$ MBPR, or that we

could use grammar-based reconstruction as in [39] on top of $\exists$ MBPR. On the other hand, our specifications support the use of uninterpreted symbols, in particular uninterpreted functions and predicates. This allows us to express e.g. the workshop problem (see Example 2), which is not directly supported by SyGuS. However, using higher-order logic, uninterpreted functions can be encoded as higher-order variables, and with this encoding cvc5 solves the workshop problem (and some other problems, see Section VI). Finally, neither SyGuS nor the deductive synthesis methods support partial function synthesis.

We provide some examples where the partial function synthesis problem has no solutions. The study of unrealizability [34] dually establishes methods to show that there are no solutions to synthesis problems.

Projections: The literature on interpolation, including [17], develops algorithms based on proof objects. Symbol elimination in saturation [28] works by deriving clauses that are consequences of $\exists u. \neg \Phi$. Negations of these clauses can be used as projections. The approach of [28] relies on symbol orderings to eliminate symbols through rewriting, but does not offer guarantees of complete symbol elimination. Deductive synthesis with uncomputables [21] blocks inferences which would lead to uncomputable realizers, but also is not complete. As an example, inferences such as $f(y) \not\simeq f(a) \vdash y \not\simeq a$ with $u = \{f\}$ are not part of this methodology.⁶

Notably, MCSAT [23] relies on a dual to $\exists$ MBP, called $\forall$ MBP, in a setting where it is used for satisfiability checking of quantifier-free formulas. The starting point for $\forall$ MBP is a partial model of a subset of symbols that cannot be completed to a full model of all symbols. A combination of $\forall$ MBP and $\exists$ MBP for quantifier solving for polynomial arithmetic is developed in [35], while $\exists$ MBP with cores and strategies are used in [7], [9], [33]

We note that stand-alone projection is not necessarily the most efficient method for solving quantifiers. Templates have recently been used successfully for non-linear arithmetic [11], and Z3’s Horn clause solver uses information from several projection invocations to compute more general projections [30].

Implied equalities: Bromberger and Weidenbach [10] present a method for extracting all implied equalities from a set of linear inequalities over the reals. It is used when solving over LIA. Z3 [8] uses an incomplete heuristic to detect implied equalities during search. The QEL algorithm [14] uses equalities that are consistent with an interpretation, but not necessarily implied by a current assignment to literals, to simplify quantifier elimination problems.

VIII. CONCLUSIONS

This paper formally defines and studies the problem of synthesis of total and partial functions in the presence of uncomputable symbols. To solve these problems, we introduced a modular algorithm for synthesis, relying on quantifier projections. We summarized requirements on the projections

⁵We used Vampire v4.9, commit dde88d57f, in the configuration `--decode dis+32_1:1_tgt=off:qa=synthesis:ep=off:alasc a=off:drc=ordering:bd=off:nm=0:sos=on:ss=included:si=on:rawr=on:rtra=on:proof=off:mss=off_600`, and cvc5 v1.1.2 in the default configuration. For cvc5, we used SyGuS translations of the problems, encoding uncomputables as non-input variables.

⁶We note that such an inference could be supported by a suitable variation of unification with abstraction [38].

and presented algorithms computing existential projection with witnesses for EUFLIA, and universal projection internally relying on existential projection. Further, we described a method to circumvent quantifier solving for uniquely defined functions in EUFLRA. Finally, we implemented a prototype demonstrating that the proposed methods work in practice.

Our results prompt future work in multiple directions, such as extending synthesis to functions with different inputs, or extending the specification with quantified background axioms to support reasoning with user-defined theories. Moreover, developing projections for other theories would allow synthesis in those theories by plugging them into our algorithm. Last but not least, our unique solution approach could be extended to infer all implied equalities.

Acknowledgments. We thank Kuldeep Meel for suggesting to examine partial function synthesis with unique realizers. Diego Olivier Fernandez Pons inspired us to explore synthesis with Z3. We also thank Mikoláš Janota for discussions of examples. Petra Hozzová was funded by the European Union under the project ROBOPROX (reg. no. CZ.02.01.01/00/22_008/0004590).

REFERENCES

- [1] S. Akshay, Supratik Chakraborty, and Shetal Shah. Tractable representations for boolean functional synthesis. *Ann. Math. Artif. Intell.*, 92(5):1051–1096, 2024.
- [2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emmina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.
- [3] Rajeev Alur, Dana Fisman, Saswat Padhi, Andrew Reynolds, Rishabh Singh, and Abhishek Udupa. SyGuS-Comp 2019. <https://sygus.org/comp/2019/>, 2019.
- [4] 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 *Proc. of TACAS*, pages 415–442, 2022.
- [5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [6] Nikolaj Bjørner, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Model based interpolation for uninterpreted functions and integer linear arithmetic. EasyChair Preprint no. 10000, EasyChair, 2023.
- [7] Nikolaj S. Bjørner and Mikoláš Janota. Playing with quantified satisfiability. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.
- [8] Nikolaj S. Bjørner and Lev Nachmanson. Arithmetic solving in Z3. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 26–41. Springer, 2024.
- [9] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Christophe Vauthier. QSMA: A new algorithm for quantified satisfiability modulo theory and assignment. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2023.
- [10] Martin Bromberger and Christoph Weidenbach. Fast cube tests for LIA constraint solving. In *IJCAR*, 2016.
- [11] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Mehrdad Karrabi, Harshit J Motwani, Maximilian Seeliger, and Đorđe Žikelić. Quantified Linear and Polynomial Arithmetic Satisfiability via Template-based Skolemization. In *AAAI*, 2025.
- [12] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, May 2008.
- [13] Azadeh Farzan and Zachary Kincaid. Linear arithmetic satisfiability via strategy improvement. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 735–743. IJCAI/AAAI Press, 2016.
- [14] Isabel Garcia-Contreras, Hari Govind V. K., Sharon Shoham, and Arie Gurfinkel. Fast approximations of quantifier elimination. In Constantin Enea and Akash 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 64–86. Springer, 2023.
- [15] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-Free Programs. In *Proc. of PLDI*, page 62–73, 2011.
- [16] Sumit Gulwani and Madan Musuvathi. Cover algorithms and their combination. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2008.
- [17] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In Hongseok Yang, editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2011.
- [18] Márton Hajdu, Petra Hozzová, Laura Kovács, Andrei Voronkov, Eva Maria Wagner, and Richard Steven Žilínčík. Synthesis Benchmarks for Automated Reasoning. <https://arxiv.org/abs/2507.19827>, 2025. To appear in proceedings of CICM 2025.
- [19] Petra Hozzová. Integrating Answer Literals with AVATAR for Program Synthesis. In Laura Kovács and Michael Rawson, editors, *Proc. of the 7th and 8th Vampire Workshop, volume 99 of EPiC Series in Computing*, pages 13–20. EasyChair, 2024.
- [20] Petra Hozzová and Nikolaj Bjørner. Online material for Synthesiz3 This! <https://doi.org/10.5281/zenodo.16436706> and <https://github.com/Z3Prover/doc/tree/master/synthesiz3>, 2025.
- [21] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In Brigitte Pientka and Cesare Tinelli, editors, *Proc. of CADE*, volume 14132 of *LNCS*, pages 307–324, Cham, 2023. Springer.
- [22] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling Enumerative and Deductive Program Synthesis. In *Proc. of PLDI*, PLDI 2020, page 1159–1174, New York, NY, USA, 2020. Association for Computing Machinery.
- [23] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.
- [24] A. S. Kahr, Edward F. Moore, and Hao Wang. Entscheidungsproblem reduced to the $\forall\exists\forall$ case. *Journal of Symbolic Logic*, 27(2):225–225, 1962.
- [25] Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *FMCAD*, pages 89–96, 09 2015.
- [26] Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design, FMCAD '15*, page 89–96, Austin, Texas, 2015. FMCAD Inc.
- [27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.

- [28] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2009.
- [29] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *Proc. of CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
- [30] Hari Govind Vadiramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. *Formal Methods Syst. Des.*, 63(1):81–109, 2024.
- [31] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In Benjamin G. Zorn and Alex Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 316–329. ACM, 2010.
- [32] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.
- [33] Charlie Murphy and Zachary Kincaid. Quantified linear arithmetic satisfiability via fine-grained strategy improvement. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 89–109. Springer, 2024.
- [34] Shaan Nagy, Jinwoo Kim, Thomas W. Reps, and Loris D’Antoni. Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. *Proc. ACM Program. Lang.*, 8(OOPSLA2):113–139, 2024.
- [35] Jasper Nalbach and Gereon Kremer. Extensions of the cylindrical algebraic covering method for quantifiers. *CoRR*, abs/2411.03070, 2024.
- [36] Markus N. Rabe, Leander Tentrup, Cameron Rasmussen, and Sanjit A. Seshia. Understanding and extending incremental determinization for 2qbf. In Hana Chockler and Georg Weissenbacher, editors, *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 II*, volume 10982 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2018.
- [37] Giles Reger. Revisiting Question Answering in Vampire. *EPiC Series in Computing*, 53:64–74, 2018.
- [38] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning. In *Proc. of TACAS*, pages 3–22, 2018.
- [39] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.*, 55(2):73–102, 2019.
- [40] Tanel Tammet. Completeness of resolution for definite answers. *J. Log. Comput.*, 5(4):449–471, 1995.
- [41] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program Synthesis Using Dual Interpretation. In *Proc. of CADE*, pages 482–497, 2015.
- [42] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In Robert Nieuwenhuis, editor, *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.

Guiding Likely Invariant Synthesis on Distributed Systems with Large Language Models

Yuan Xia, Aabha Shailesh Pingle, Deepayan Sur, Srivatsan Ravi, Mukund Raghothaman, Jyotirmoy V. Deshmukh

University of Southern California

Los Angeles, USA

{yuanxia, apingle, deepayan, srivatsr, raghotha, jdeshmuk}@usc.edu

Abstract—Deductive verification is an approach where the user expends effort in constructing a program invariant manually, or learns this invariant from program traces, and subsequently uses the program invariant for safety verification tasks. Most existing invariant synthesis tools still rely on model checking methods to certify invariants, use enumerative search to identify and refine candidate invariants, and user insights to start with a set of candidate predicates that may appear in the final (learned) invariant. On the other hand, pre-trained large language models (LLMs) have recently attracted considerable attention in various tasks related to generating program code (e.g., from natural language). There have been recent efforts at applying LLMs to the program verification context. In this paper, we investigate the capabilities of LLMs to synthesize program invariants by creating a specialized set of prompts. When LLM invariant synthesis with direct prompting fails, we introduce two revision frameworks for incorporating LLM calls. In the first framework PSyn, the LLM suggests atomic predicates based on counterexamples for reconstructing the invariants. The second framework ISyn adopts the LLM to generate invariants by itself within a revision process. We tested our methods on 8 distributed systems modeled in Promela and invariants are verified with Spin. Our results show that the state-of-the-art model GPT-o3, when it is used to generate invariants directly, is less effective than the symbolic invariant synthesis method RunVS. However, our integrated approach PSyn, which employs the LLM as a predicate prompter, significantly outperforms GPT-o3 and ISyn and has comparable performance with RunVS. This integrated technique also produces higher-quality invariants in general. Because we focus on runtime monitoring frameworks, we primarily consider system traces and hence likely invariants. However, our revision frameworks can also learn true invariants by using model checkers in the revision frameworks.

Index Terms—Invariant Synthesis, Distributed Systems, Large Language Models

I. INTRODUCTION

Formal verification of distributed computing systems is a computationally challenging problem, as the number of system states grows exponentially both in the number of program variables, and the number of processes participating in the distributed computation. This state-space explosion problem has long been a roadblock for automatic verification techniques like model checking, though approaches like partial order reductions, symmetric reductions, symbolic methods with SMT solvers, and context-bounding have sought to ameliorate this problem. An alternative approach to exhaustive exploration of the state-space is deductive verification, where the user proposes an invariant expression over the (global) program

variables, which if valid, can help simplify proofs of system safety to a simple validity check of a propositional formula.

As manually deriving invariants is difficult, there has been considerable effort to automatically synthesize program invariants. There are two primary methods for this: deductive and automatic methods. Deductive methods involve manual or semi-manual proofs aided by theorem provers such as Coq [1] and Isabelle [2], and involve interaction with a proof system with explicit axiomatization of the program syntax and semantics. Deductive methods [3], [4] are able to capture complex invariants, but require significant human labor and deep domain knowledge. Automatic methods rely on tools such as model checkers or dynamic invariant detectors to learn invariants.

Model-checker-assisted invariant synthesis methods iteratively refine candidate invariants based on counterexamples generated by model checkers. Methods based on ICE-learning [5], [6] use decision trees to classify positive examples obtained from program traces from negative examples obtained from user-provided safety properties and implication counterexamples obtained from program transitions to enforce inductiveness of the invariant. Invariant verification is performed using the Z3 SMT solver. DistAI [7] leverages Ivy [8] as a model checker to ensure inductiveness of the invariant. While these approaches can give deterministic guarantees on the correctness of the invariant, they may face computational challenges in validating the invariant for large or infinite-state systems.

Dynamic invariant synthesis techniques infer invariants from just execution traces. Tools such as Daikon [9] collect runtime observations and detect likely invariants from them. DIDUCE [10] tracks certain types of invariants of Java programs. While dynamic methods may overlook rare edge cases and produce overly broad or imprecise invariants, they are generally faster and more scalable in predicting invariants from system traces. Most automatic invariant synthesis methods require user input in the form of annotations (e.g., in the form of assertions), identification of program variables that should be included in the invariant, and initial sets of predicates. In this paper, we will focus on automatic methods, while trying to both alleviate the burden imposed on users and scaling the process of validating the synthesized invariant.

A step in this direction was taken by our prior work [11] which developed the RunVS technique for dynamic invariant

learning. RunVS learns decision tree-based invariants to separate positive examples (obtained from execution traces) from speculative negative examples. The approach uses a runtime monitoring technique to validate the learned invariant. As it does not provide deterministic guarantees on the learned invariant, we call its output a *likely invariant*. However, there are a number of limitations with RunVS:

- 1) RunVS requires the user to provide a list of global variables that may be included in the invariant.
- 2) RunVS requires a grammar for the predicate augmentation.
- 3) In many distributed systems, some global variables are indexed by the processes that update or read from them, and invariants can be quantified expressions such as $\forall i \exists j : P(x_i, y_j) \implies Q(x_i)$, where P and Q are predicates involving two variables and a single variable, respectively [7].

To address these challenges, we propose two frameworks that integrate Large Language Models (LLMs) into the invariant synthesizer RunVS. When we prompt the LLM with the source code for a distributed protocol along with other carefully crafted queries, the LLM can suggest predicates, reducing the space of predicates to be explored. When prompted with counterexamples and some of the previous context, it can revise invariants. In RunVS, we iteratively learn the invariant by alternating invariant inference (from examples) and runtime monitoring to generate counterexamples. Any counterexamples found are used to refine the invariant. We use the same basic framework, but use LLM-guidance at several steps during this process.

Contributions. To summarize, this paper makes the following contributions:

- (i) We investigate prompt design for LLMs to discover and refine predicates that may appear in a program invariant.
- (ii) We mitigate the problem of exploring a large space of predicates by combining LLMs with the RunVS invariant revision framework.
- (iii) We boost LLM’s predicate prediction capability with chain of thought reasoning.
- (iv) We extend our previous RunVS approach to synthesize quantified invariants by using the LLM to replacing quantifier-free expressions with inferred quantifiers.
- (v) We demonstrate the usefulness and scalability of our framework on distributed systems modeled in Promela [12].

II. PRELIMINARIES

In this section, we formalize the terminology needed to explain our approaches. Besides, we will present the problem statement in this section.

A. Program structure and modeling assumptions

Program variables, program states, program execution traces, and invariants are all fundamental concepts in program synthesis. We give their definitions in this section so that we can precisely establish our problem statement.

Definition 1 (Variables, States). A type t refers to a finite or an infinite set of values. A program variable v of type t represents a symbolic name that contains values from t , denoted by $\text{type}(v) = t$. Let V represent the complete set of program variables. A valuation $\nu : V \rightarrow \bigcup_{v \in V} \text{type}(v)$ is a function that assigns to each variable $v \in V$ a concrete value $\nu(v) \in \text{type}(v)$. The program state at any execution point is denoted by the tuple $\nu(V)$, representing the collective valuation of all variables in V .

Definition 2 (Labelled Transition System (LTS)). A labelled transition system for a concurrent program is a tuple (S, L, T, Init) , where:

- S is a set of system states,
- L is a set of labels corresponding to program statements,
- $T \subseteq S \times L \times S$ is the transition relation,
- $\text{Init} \subseteq S$ is a set of initial states.

Definition 3 (Reachable States). For a program $\mathcal{P}$ modeled as an LTS, its reachable states R are all states that can be reached from initial states s_0 , $s_0 \in S$. $\text{Reach}(\mathcal{P}, \text{Init})$ is the minimal set R satisfying:

- 1) $s_0 \in \text{Init} \implies s \in R$ and,
- 2) $s \in R \wedge (s, s') \in T \implies s' \in R$.

Definition 4 (Program Trace). A program trace $\sigma = (s_0, s_1, \dots, s_k)$ is a finite sequence of states where: $s_0 \in \text{Init}$, and for all $j \in [1, k-1]$, $(s_{j-1}, s_j) \in T$.

Definition 5 (Monitor). A monitor observes a finite prefix $\sigma = (s_0, \dots, s_k)$ of a program trace and evaluates whether σ satisfies a property ϕ .

Runtime monitoring refers to monitoring the system’s states during its execution and detecting whether the behaviors align with predefined specifications. Runtime monitoring is a popular approach for dynamic verification [13]. The *offline monitoring* [14] collects runtime information during the system’s execution and stores it for later analysis. The analysis is performed offline once the system has completed execution. In contrast, *online monitoring* [14] refers to the simultaneous observation and analysis of system states as it is running. The concurrent monitor enables early detection and response, which is essential for critical systems requiring immediate corrective actions.

Definition 6 (Invariants). An invariant I is a Boolean-valued expression over program variables that holds for all reachable program states, formally defined as follows:

$$\forall s \in \text{Reach}(\mathcal{P}, \text{Init}), s \models I.$$

Remark. Likely invariants are hypothesized invariants that hold true for all states observed up to a given point.

B. Basics of LLM-based analysis

Generative Large Language Models (LLMs) are transformer-based neural networks pre-trained on large text/code corpora to generate human-like outputs. LLMs leverage self-attention mechanisms to capture long-range

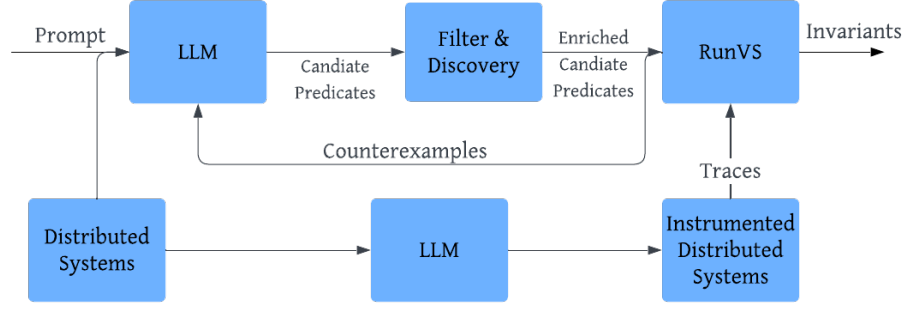


Fig. 1: Predicate LLM Synthesis Framework

dependencies in sequences. With fine-tuning, they are adaptable to domain-specific tasks, e.g., code synthesis [15]. In this work, LLMs act as assistants for invariant synthesis, helping revise predicates or invariants. Few-shot prompting (learning) is a form of prompt engineering in generative AI, where a few examples are provided in a prompt to perform a task.

Zero-shot prompting refers to the method of no examples given, and one-shot prompting means a single example crafted in a prompt to achieve a desired output. One-shot prompting is particularly useful when collecting a large amount of training data is impractical. We will use one-shot prompting to perform the task of invariant synthesis, denoted as one-shot invariant synthesis, as our baseline experiments.

Problem statement. Let $\mathcal{P}$ be a distributed program. Let $\text{Reach}(\mathcal{P}, \text{Init})$ be the set of reachable states of $\mathcal{P}$. Let G denote the grammar to specify (a possibly infinite) set of Boolean-valued expressions over program variables, and let $\mathcal{L}(G)$ denote this set. The objective of this paper is to design a runtime algorithm able to learn a program likely invariant ϕ without dependency on users by integrating LLMs. The invariant has the following properties:

- 1) Soundness of a likely invariant:

$$((s \in \text{Reach}(\mathcal{P}, \text{Init})) \Rightarrow (s \in \phi)) \quad (1)$$

- 2) Tightness of a likely invariant:

$$\phi = \arg \min_{\phi \in \mathcal{L}(G)} |\{s \mid s \notin \text{Reach}(\mathcal{P}, \text{Init}) \wedge s \in \phi\}| \quad (2)$$

III. OVERVIEW

In this section, we will first introduce our code instrumentation technique for tracking state changes of a distributed system simulated by Spin [16]. We then present two runtime monitoring frameworks integrating revision guidance for invariant synthesis from LLMs.

LLM-aided Code Instrumentation. Spin can monitor the state changes of a system modeled in Promela by capturing the system states after each system transition. However, its default approach treats an atomic block consisting of multiple statements, not as a single, simultaneous event, but as individual statements. Thus, Spin does not record global states after

atomic blocks. In other words, atomicity is not held in Spin. The traces do not accurately show the states captured “before” and “after” snapshots of atomic blocks. We propose a refined instrumentation strategy that encodes a Promela system into an instrumented Promela system that is trackable by Spin, as shown in Figures 1 and 2. The implementation leverages an LLM to address this limitation. Our technique automatically inserts instrumentation code after every assignment outside atomic blocks while only considering one state update at the end of every atomic block when assignment statements are inside. The LLM produces updated Promela code given a sequence of required instrumentation tasks and constraints in the natural language specification. This method offers an automatic solution for capturing intermediate state transitions without sacrificing the atomic semantics of systems modeled by Spin. We will adopt Peterson [16] Promela program as a demonstration example throughout all prompting techniques. Peterson is a tiny concurrent algorithm that implements mutual exclusion for two concurrent processes.

In this paper, we introduce two frameworks for runtime invariant synthesis that integrate revision guidance from LLMs. Both frameworks rely on iterative refinement within runtime monitoring. The frameworks are supposed to detect incorrect invariants by observing system traces and refining them based on feedback, which is system states in observed traces that do not satisfy candidate invariants, which are denoted as counterexamples. However, two frameworks are distinguished regarding how an LLM is employed and what task the LLM is assigned.

Predicate LLM Synthesizer Framework. As shown in Figure 1, the first framework treats the LLM as a predicate hinter, which is considered a component responsible for supplying promising atomic predicates that are precise and potentially exist in the system invariants. Example prompts are presented in the section IV. We iteratively prompt the LLM with stored predicates from the previous iteration, the falsified invariant, and counterexamples observed from the system’s runtime monitoring. Counterexamples are the system states that make the candidate invariant fail. We maintain a certain length of chat history as part of the context in LLM’s input. This approach ensures that the LLM’s context remains focused on

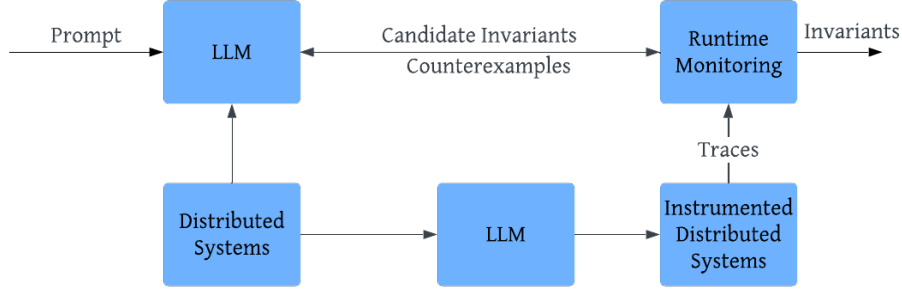


Fig. 2: Invariant LLM Synthesis Framework

relevant information, preventing it from exceeding the LLM’s context window limit and maintaining a reasonable prompt length.

The new atomic predicates proposed by the LLM might augment the stored predicates. The stored predicates are increasingly augmented and refined over iterations. RunVS will construct an invariant that is sound and precise based on the input of predicates. Candidate invariants are expected to hold for every system trace. If the invariant fails to hold, the system extracts the state as a new counterexample, updates the counterexample set, and prompts the LLM again. This Predicate Prompter framework effectively offloads the challenging task of demanding potentially useful predicates from the user end to an LLM while keeping the invariant generation and monitoring rigorous through formal synthesis. We further filter out invalid predicates and enrich valid predicates, which is explained in the next section. Unlimited positive states will eventually exceed the context window of the LLM. Therefore, we focus on the validated predicates and a certain number of counterexamples during prompting.

Invariant LLM Synthesizer Framework. The self-revised LLM framework, as shown in Figure 2, adopts an LLM to generate entire invariants and iteratively refining them. In this framework, the LLM produces complete candidate invariants rather than atomic predicates. Example prompts are shown in section V. During the monitoring, once counterexamples are observed and a revision is triggered, these counterexamples are fed back to the LLM and prompted to revise the proposed invariant regarding the detected error states. Instead of incrementally adding predicates, this method explores the feasibility of the LLM’s capability to generate and revise invariants by itself iteratively. In this framework, an LLM is guided by prior counterexamples and the information of partial successes of the candidate invariant in the previous iteration. While potentially more flexible, the self-revised LLM framework also leads to more concerns, e.g., the soundness and tightness guarantee of the guessed invariants.

Comparisons. The predicate LLM synthesizer framework builds invariants formally based on predicates, which reduces LLM’s burden of struggling with invariant expression format correctness and logic correctness. We allow an LLM integra-

tion but still rely on RunVS to construct likely invariants, which are sound and precise in terms of the observed states. The invariant LLM synthesizer framework aims for a more holistic view, which may converge quickly if the LLM can combine multiple relevant constraints effectively. However, it can also require more sophisticated prompt engineering to avoid losing key information and giving too general invariants. Both frameworks consider incorporating LLM-assisted invariant generation with counterexample-guided refinement.

IV. PREDICATE LLM SYNTHESIZER

As shown in Algorithm 1, PSyn collects counterexamples and samples a certain number of negative states in each iteration at Line 7, and ensures the recall is 1 and the precision is higher than a predefined threshold at Line 8. Otherwise, a revision will be triggered to learn a sound and more precise invariant. As Line 10, our predicate LLM synthesizer is fed with predicates, counterexamples, and the previous candidate invariant. It is responsible for drafting a new set of candidate predicates P in each required revision.

For an ablation study, we have included three types of prompting techniques, i.e., zero-shot, one-shot, and chain-of-thought (CoT) prompting, for testing and improving the inference capability of the LLM. For zero-shot prompting, in the first iteration, we supply the LLM with the full list of global variables, a Promela program, and the acceptable operators, e.g., $+$, $-$, $==$, $<=$, and, or, true. For one-shot learning, we adopt the Peterson promela program along with an answer of predicates as one demonstration example, which is inserted into the prompt body. This biases the model towards catching the correct syntax and semantics of predicates from a given code. For CoT prompting, we prepare a reasoning template instructing the model to “think aloud” of predicates from the following five aspects:

- 1) Assertions embedded in the Promela model,
- 2) Branch conditions, e.g., guards in *if*, *do*, and *goto* statements,
- 3) Template predicates obtained from the above information,
- 4) Analysis of observed system states combined with variable types to infer variable ranges and replace template predicates with concrete values, e.g., $0 \leq ncrit \leq 1$,

Algorithm 1: PSyn

input : Distributed/Concurrent program $\mathcal{P}$
The number of points randomly sampled l
The precision threshold δ
Maximum depth of decision tree k
 $\frac{|\text{speculated}|}{|\text{reached}|}$ threshold α
Total runs to monitor MaxTraces
LLMs generate or user input atoms
Max runs to confirm final invariant
MonitorBudget
output: likelyInv ϕ

```
1 reached  $\leftarrow \{\}$ , speculated  $\leftarrow \{\}$ ,  $\phi \leftarrow \text{false}$ ,  $m \leftarrow 1$ ,  
   $n \leftarrow 0$   
2 repeat  
3    $T \leftarrow \text{SampleTrace}(\mathcal{P})$  // sample a random trace  
4    $\text{ce} \leftarrow \{s \mid s \in T \cap \llbracket \neg\phi \rrbracket\}$  // reached state in  $\neg\phi$   
5    $\text{ce} \leftarrow \text{ce} \cup \{s \mid s \in T \cap \text{speculated}\}$  // reached  
     state assumed unreachable  
6    $\text{reached} \leftarrow \text{reached} \cup \text{ce}$  // update reached states  
7    $\text{speculated} \leftarrow (\text{speculated} \setminus \text{ce}) \cup \text{randomSample}(S \setminus$   
      $((\text{speculated} \setminus \text{ce}) \cup \text{reached}), \ell)$   
8     s.t.  $\frac{|\text{speculated}|}{|\text{reached}|} < \alpha$   
9   if  $\text{ce} \neq \emptyset$  then  
10    predicates  $\leftarrow$   
      LargeLanguageModel(atoms, ce,  $\phi$ )  
11    atoms  $\leftarrow$   
      PredicateDiscovery(BreakDown(predicates))  
12     $\phi \leftarrow$   
      Learner(atoms, reached, speculated,  $d, k, \delta$ )  
       $n \leftarrow 0$   
13  else  $n \leftarrow n + 1$   
14   $m \leftarrow m + 1$   
15 until  $m \geq \text{MaxTraces} \vee n \geq \text{MonitorBudget}$   
16 return  $\phi$ 
```

- 5) Counterexamples used to find new satisfying predicates to update the current predicate set.

We include only the validated predicates and the latest counterexamples so that the prompt length stays below the model's context limit. The model returns a list of candidate predicates $P = \{p_1, \dots, p_k\}$. p_i is a Boolean expression. We apply predicate breakdown at Line 8 to enrich more predicates, where we parse the predicates into sub-clauses that can serve as atomic predicates. To ensure the format correctness of the generated predicates, we use a predicate filter function to filter out predicates in a wrong syntax. Besides, PredicateDiscovery at Line 8 enriches more concrete predicates by extracting the predicate patterns and replacing them with concrete values detected from system traces. The predicates coming from PredicateDiscovery will be the final predicates fed into the decision tree invariant learner Learner.

Algorithm 2: ISyn

input : Distributed/Concurrent program $\mathcal{P}$
The number of points randomly sampled l
The precision threshold δ
Max runs to confirm final invariant
MonitorBudget
output: Revised invariant ϕ

```
1 repeat  
2    $T \leftarrow \text{SampleTrace}(\mathcal{P})$  // sample a random trace  
3    $\text{ce} \leftarrow \{s \mid s \in T \cap \llbracket \neg\phi \rrbracket\}$  // reached state in  $\neg\phi$   
4    $\text{ce} \leftarrow \text{ce} \cup \{s \mid s \in T \cap \text{speculated}\}$  // reached  
     state assumed unreachable  
5    $\text{reached} \leftarrow \text{reached} \cup \text{ce}$  // update reached states  
6    $\text{speculated} \leftarrow (\text{speculated} \setminus \text{ce}) \cup \text{randomSample}(S \setminus$   
      $((\text{speculated} \setminus \text{ce}) \cup \text{reached}), \ell)$   
7     s.t.  $\frac{|\text{speculated}|}{|\text{reached}|} < \alpha$   
8   if  $\text{ce} \neq \emptyset$  then  
9      $\phi \leftarrow \text{LargeLanguageModel}(\text{ce}, \phi)$   
10     $n \leftarrow 0$   
11  else  $n \leftarrow n + 1$   
12   $m \leftarrow m + 1$   
13 until  $m \geq \text{MaxTraces} \vee n \geq \text{MonitorBudget}$   
14 return  $\phi$ 
```

A. Expansion to LLM-Guided Quantifier Synthesis

We also integrated an LLM for the expansion to quantified invariants. LLM is used to automate two critical tasks, i.e., detecting quantifiable variables and generalization to quantified invariants. Firstly, the LLM analyzes the Promela source code with a structured prompt to recognize quantifiable variables. LLM is able to identify the quantifiable variables and their names, types, and the corresponding domains of the variables. For instance of the distributed lock system [16], the LLM identified one quantifiable global variable $\text{current_holder} \in \{-1, 0, \dots, N-1\}$. Secondly, given a candidate invariant proposed by the symbolic synthesizer RunVS, the LLM synthesizes a quantified formula by replacing enumerative clauses, e.g., $\text{current_holder} = 0 \vee \text{current_holder} = 1 \vee \dots$, with semantically equivalent quantified expressions, e.g., the existential quantifier $\exists \text{pid} (0 \leq \text{pid} < N \wedge \text{current_holder} = \text{pid})$. For the distributed lock system, the LLM produces $(\text{lock_available} \wedge \text{current_holder} = -1) \vee (\neg \text{lock_available} \wedge \exists \text{pid} (0 \leq \text{pid} < N \wedge \text{current_holder} = \text{pid}))$. This generalization preserves the logic of invariants and tends to generate more precise invariants. This step enables reasoning across arbitrary system scales by eliminating explicit dependence on N and enables the length optimization for human understanding.

V. INVARIANT LLM SYNTHESIZER

Our second framework, ISyn, assigns the invariant generation and revision task to an LLM to replace the decision tree learner in RunVS. Unlike PSyn, this LLM treats an invariant as a single boolean expression that is regenerated after every

revision. The framework is shown in Algorithm 2. We still adopt structured prompts and the ablation study for better instruction. We included the previous three chat histories and subsampled a fixed number of counterexamples to prompt the LLM. We only prompt the LLM with the latest counterexamples to avoid exceeding the context window limit, shown at Line 9. Before accepting the LLM-generated invariant, we run a syntactic check to ensure the format correctness of the invariant. We employed zero-shot learning, one-shot learning, and CoT prompting for the ablation study of the prompting in the second framework. For the zero-shot prompt, we directly prompt LLM to produce an invariant that always holds given the Promela program. With a one-shot prompt we provide the Peterson Promela code and its corresponding correct invariant as a demonstration example. For CoT prompting, besides the aspects described in the previous section, we design thinking steps that involve invariant inference and invariant revision.

Model-checker validation. Our frameworks can accept feedback from a model checker to prompt the LLM. Spin is able to verify an invariant if it is sound. If Spin confirms the soundness, the loop terminates. Otherwise, a reproducible trace with the violated state is returned. The next revision prompt includes the verification result, the error state, and the last monitored invariant. Our LLM synthesizers are able to revise invariants or predicates based on the model checker’s verification results.

VI. EXPERIMENTAL EVALUATION

A. Benchmarks and Measurements

To investigate our research goal, experiments were designed to address the following three research questions:

- **RQ1:** Do different prompting techniques affect the overall performance of runtime invariant generation?
- **RQ2:** Do PSyn and ISyn generate the likely invariants of higher quality than those generated by GPT-o3 and RunVS by runtime monitoring of these real-world distributed systems?
- **RQ3:** How PSyn and ISyn efficiently and effectively learn likely invariants on large-scale distributed systems?

We employ *gpt-3.5-turbo-0125* [17] as the component within our PSyn and ISyn frameworks. We configured the LLM API with temperature $T = 1$ to balance deterministic output with randomness and diversity. Counterexample sampling was constrained to 5 examples per iteration, optimizing the trade-off between context window utilization and informativeness. We adopt RunVS [11] as our symbolic invariant synthesizer, which is the only tool able to learn diverse invariants on Spin systems within a runtime monitoring framework. As baseline experiments, we evaluate invariants that are purely generated by RunVS within 500 traces. Besides, we adopt GPT-o3 [17] among the state-of-the-art models. We perform one-shot invariant synthesis by GPT-o3 with the context of program code but without a revision framework.

We adopt three metrics for evaluating the quality of learned invariants. The first two are soundness and tightness, whose

definitions are defined in the problem statement. Soundness is checked by Spin with the corresponding system. Tightness is evaluated by counting the assignments of a boolean expression Z3 SMT solver. Tighter invariants indicate that the invariants are more concise. The third one is safety, which can be considered as proof of invariant usefulness. Safety is evaluated by implication checking between the invariant i and the safety property ϕ such that $i \rightarrow \phi$.

Our benchmarks include 8 distributed systems modeled in Promela, drawn from resources related to Spin and other distributed systems literature. Our frameworks PSyn and ISyn are general to other languages. Promela is chosen since it represents interesting finite-state, infinite-state, asynchronous message-passing-based distributed systems and is the input to the Spin model checker, which gives us fine-grained control over trace generation.

B. Prompt Engineering Ablation Study

In Tables I and II, we compared the performance of three prompting techniques, i.e., direct prompting, one-shot prompting, and chain-of-thought prompting [18]. In each case study, if a synthesized likely invariant is verified by Spin or implies the safety property, it is indicated by ✓ in tables. The tightness is evaluated by calculating the model counts by Z3 solver [19]. Since we adopted Peterson as our demonstration example in our 1-shot and CoT prompting, Peterson was not experimented with using those two prompting techniques. We can see CoT tends to learn a sound and more precise invariant since LLM adopts reasoning to give more diverse predicates. Besides, one-shot and CoT can give a more useful invariant since their generated invariants mostly imply the safety property of the corresponding system. CoT especially helps ISyn to break down the whole reasoning steps of invariant synthesis compared to the direct prompting’s useless invariants. However, we can see LLM prompting techniques may fail to predict all potential predicates that can generate a sound invariant to predict all reachable states for some system, i.e., producer-consumer, which involves complex mathematical equations to represent its reachable states.

C. Invariant Quality Evaluation

Based on the ablation study of different prompting techniques, PSyn and ISyn with CoT prompting perform relatively better than other two techniques. We adopt CoT PSyn and ISyn to compare with RunVS and GPT-o3. We can see PSyn under chain-of-thought prompting has comparable performance with RunVS symbolic tool with user-provided predicates. However, its usefulness declines when predicates demand intricate combinations of mathematical expressions. Even users must already have a comprehensive knowledge of the system to formulate such predicates. ISyn-generated invariants are more general than PSyn’s, but compared to GPT-o3’s, more invariants imply safety properties, which indicate the usefulness of generated likely invariants. Besides, for deeper analysis of ISyn results, we conduct root cause analysis over ISyn-generated invariants in Table IV.

System	Tightness			Soundness			Safety		
	0-shot	1-shot	CoT	0-shot	1-shot	CoT	0-shot	1-shot	CoT
Peterson [16]	10	—	—	✓	—	—	✓	—	—
Bakery [16]	2	2	2	✓	✓	✓	✓	✓	✓
Alternating Bit Protocol [16]	300	300	300	✓	✓	✓	✓	✓	✓
Leader Election [16]	2	2	2	✓	✓	✓	✓	✓	✓
UPPAAL Train/Gate [16]	202	16	14	✓	✓	✓	✗	✓	✓
Distributed Lock Server [20]	6	6	6	✓	✓	✓	✓	✓	✓
Producer Consumer [21]	505000	10200	5050	✗	✗	✗	✗	✓	✓
Smart Contract(Ethereum Transactions) [22]	7200	9700	9500	✓	✓	✓	✓	✓	✓

TABLE I: **Likely Invariant Quality Evaluation** of PSyn with different prompting techniques

System	Tightness			Soundness			Safety		
	0-shot	1-shot	CoT	0-shot	1-shot	CoT	0-shot	1-shot	CoT
Peterson [16]	16	—	—	✓	—	—	✓	—	—
Bakery [16]	200	2	199	✓	✓	✓	✗	✓	✗
Alternating Bit Protocol [16]	400	300	301	✓	✓	✓	✗	✓	✓
Leader Election [16]	3	3	3	✓	✓	✓	✓	✓	✓
UPPAAL Train/Gate [16]	300	400	399	✓	✓	✓	✓	✗	✓
Distributed Lock Server [20]	11	12	7	✓	✓	✓	✗	✗	✓
Producer Consumer [21]	250000	505000	1000000	✗	✗	✓	✗	✗	✗
Smart Contract(Ethereum Transactions) [22]	10000	10000	10000	✓	✓	✓	✓	✓	✓

TABLE II: **Likely Invariant Quality Evaluation** of ISyn with different prompting techniques

System	Tightness				Soundness				Safety			
	PSyn	ISyn	RunVS	GPT o3	PSyn	ISyn	RunVS	GPT o3	PSyn	ISyn	RunVS	GPT o3
Peterson [16]	10	16	30	16	✓	✓	✓	✓	✓	✓	✓	✓
Bakery [16]	2	2	2	2	✓	✓	✓	✓	✓	✓	✓	✓
Producer Consumer [21]	5050	1000000	10100	1000000	✗	✓	✓	✓	✓	✗	✓	✗
Alternating Bit Protocol [16]	300	301	300	400	✓	✓	✓	✓	✓	✓	✓	✗
Leader Election [16]	2	3	2	100	✓	✓	✓	✓	✓	✓	✓	✗
UPPAAL Train/Gate [16]	14	399	300	210	✓	✓	✓	✓	✓	✓	✓	✗
Distributed Lock Server [20]	6	7	6	5	✓	✓	✓	✗	✓	✓	✓	✗
Smart Contract (Ethereum) [22]	9500	10000	10000	10000	✓	✓	✓	✓	✓	✓	✓	✓

TABLE III: **Likely Invariant Quality Evaluation** on GPT-o3, RunVS, PSyn and ISyn

Root Cause	Explanation
<i>Over-conservativeness</i>	After many counter-examples the model collapses to an overly general invariant (e.g. <code>true</code>), which trivially holds but conveys no useful information.
<i>Partial Description</i>	An invariant is too precise or incomplete.
<i>CE Violation</i>	Generated invariants are not satisfied by counterexamples.
<i>Specification Violation</i>	Generated invariant omits or contradicts constraints explicitly stated in the synthesis instructions (e.g., operator misuse or redundant information).
<i>Hallucination</i>	The LLM introduces symbolic names (variables, channels) that are <i>absent</i> from the original Promela model, yielding syntactically valid but semantically meaningless predicates.

TABLE IV: Typical root causes of incorrect LLM-generated invariants.

D. Invariant Generation Efficiency Evaluation

To benchmark the *efficiency* of PSyn and ISyn, we measured the execution time, the ratio of observed states to the total reachable states of the program (determined either manually

or through the Spin model checker), the average number of revision events, and the average longevity of final likely invariants. The outcomes for each distributed system are presented in Table V. We can see the likely invariants can be converged within a few revision events and can survive for a large number of traces. Furthermore, we show the average execution time of each LLM API call and each revision event. We can see the invariant can be revised efficiently without incurring much overhead per LLM call.

VII. VALIDITY THREATS

The performance of LLMs may be influenced by their training data. If the models were exposed to the Promela invariant corpus during pre-training, their outputs could reflect memorization rather than reasoning. To mitigate this, we included Promela programs crafted by humans to avoid a potential overlap with the LLM training dataset. Besides, we have shown the baseline LLM model’s performance at the time of the experiments. The baseline LLM is not good at invariant synthesis with a direct prompt. Besides, our evaluation focuses on classical distributed systems modeled on Spin and has

Distributed Program	LoC	Shared Vars.	No. of Reachable States (R)	$ \text{Visited} $	$\lceil f \rceil$		$\lceil r \rceil$		T_r (s)		T_{LLM} (s)	
				$ R $	PSyn	ISyn	PSyn	ISyn	PSyn	PSyn	ISyn	ISyn
Peterson [16]	20	3	16	1	400	498	1	2	0.051	26.77	1.22	
Bakery [16]	24	2	8	1	497	498	2	2	0.729	3.54	1.32	
Producer Consumer [21]	37	4	1.03M	0.00044	400	498	14	2	0.097	1.06	2.23	
Alternating Bit Protocol [16]	42	3	∞	≈ 0	400	498	1	2	0.073	1.72	2.49	
Leader Election [16]	127	3	26K	0.0024	499	498	3	2	0.053	0.59	2.93	
UPPAAL train/gate [16]	78	7	16.8M	0.000024	494	498	2	2	0.069	1.39	3.91	
Distributed Lock Server [20]	100	4	12.2K	0.015	497	498	3	2	0.054	30.47	1.14	
Smart Contract (Ethereum Transactions) [22]	962	21	∞	≈ 0	499	499	1	1	0.182	23.14	3.46	

TABLE V: **Evaluation results of PSyn and ISyn on large-scale distributed systems.** f : number of sample runs (out of 500) that the final likely invariant survives; r : average number of runs that trigger a revision; T_r : average revision time for PSyn; T_{LLM} : average LLM-interaction time per revision.

not experimented in the whole domain, e.g., systems in other languages, sequential programs. Future work should validate our frameworks on real-world systems.

Hyperparameters could affect the overall performance of the LLM. Hyperparameter tuning is not done in this work. Besides, experiments involving LLMs, e.g., GPT-3.5, are inherently non-deterministic due to API-level randomness and opaque model updates. While we report the parameters and prompts used for the experiments, exact replication may require access to fixed model versions. These threats are inherent to LLM-based approaches but do not invalidate our core findings. LLMs can aid in invariant synthesis when used properly, and structured revision frameworks have comparable performance with the symbolic solver and outperform direct prompting for invariant synthesis.

VIII. RELATED WORK

Traditional Invariant Synthesis. Prior work on invariant synthesis has focused on dynamic analysis and symbolic reasoning. Daikon [9] pioneered dynamic invariant detection by inferring likely invariants from execution traces, while DIDUCE [10] tracks certain types of invariants for Java programs. Houdini [23] starts with candidate invariants and removes invariants with the ESC/Java checker. RunVS learns decision tree-based invariants with user-given predicates. These works are more scalable but require more of the given input. Tools like ICE/ICE-DT [5], [6], which use implication counterexamples and decision trees for inductive invariant synthesis. Horn-ICE [24] extended this to handle Horn clauses for sequential and concurrent programs. These methods rely heavily on formal solvers or model checkers, limiting scalability for complex systems. DIG [25] is limited to numerical invariants. DistAI [7] and DuoAI [26] are only limited to the Ivy language and platform. These works have a certain constraints on the invariants or systems.

LLM-Based Invariant Generation. Recent studies explore LLMs for invariant synthesis. Pei et al. [27] demonstrated that LLMs can predict invariants statically, outperforming Daikon when limited to at most five execution traces. Janßen et al. [28] showed that ChatGPT generates valid loop invariants for 75 out of 106 C programs, enabling Frama-C to verify previously unprovable tasks. Chakraborty et al. [29] proposed reranking

LLM-generated invariants to reduce verification effort, achieving a median rank of 4 for correct invariants. SymbolicGPT [30] treats invariant learning as a symbolic regression task using a novel transformer-based language model. However, these approaches treat LLMs as passive generators without iterative refinement.

Besides, there are hybrid neuro-symbolic frameworks integrating LLMs with formal methods. Lemur [31] combines LLMs with automated reasoners, using LLMs to propose subgoals validated by SMT solvers. LaM4Inv [32] adopts LLMs with bounded model checking (BMC), where LLMs generate candidates and BMC provides counterexamples for revision. While Lemur leverages LLMs for invariant revision-specific subtasks, it does not fully exploit LLMs’ iterative revision capabilities during synthesis. Unlike LaM4Inv, which relies on BMC for corrections and does not fully exploit the information output from LLM, our method enables LLMs to self-correct and extracts atomic predicate templates from LLMs. Besides, LaM4Inv filters out all predicates disproved by BMC without reconstruction, which reduces the diversity and the tightness of invariants. Our work introduces two novel invariant revision frameworks. The synthesizer improves its atomic predicate suggestions based on counterexamples, enabling the LLM to reconstruct its predicates iteratively.

IX. CONCLUSION

This work establishes that structured revision frameworks significantly enhance the scalability of invariant synthesis with LLMs and also the capability of LLMs in invariant synthesis. Our experimental evaluation demonstrates that iterative refinement on atomic predicates with counterexample guidance improves invariant quality across three critical metrics: soundness, tightness, and safety. We have shown the effectiveness of chain-of-thought prompting over few-shot learning. LLM in PSyn independently revises predicates using structured feedback, akin to human debugging. By integrating LLMs into the invariant synthesis loop, rather than treating them as standalone generators, our frameworks represent a collaborative neuro-symbolic invariant synthesis. Our contribution is to integrate LLM-proposed predicates into a runtime verification loop in a structured way. To our knowledge, this is the first work to combine LLM reasoning and invariant synthesis in

distributed systems. Specifically, two frameworks are new examples of combining LLM reasoning and lightweight formal techniques, such as dynamic invariant generation and run-time monitoring on distributed systems, which increases the scalability. Future work will investigate four directions, i.e., application to other platforms (e.g., P language), exploring code- or math-specialized LLMs and fine-tuning, extension to online monitoring framework, and extension to self-debugging or self-healing systems.

Acknowledgments. This work was partially supported by the National Science Foundation through the following grants: CAREER award 2048094, FMITF-2124431, and an Amazon Faculty Research Award.

REFERENCES

- [1] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.
- [2] M. Wenzel, “Isabelle as document-oriented proof assistant,” in *International Conference on Intelligent Computer Mathematics*. Springer, 2011, pp. 244–259.
- [3] I. Dillig, T. Dillig, B. Li, and K. McMillan, “Inductive invariant generation via abductive inference,” *Acm Sigplan Notices*, vol. 48, no. 10, pp. 443–456, 2013.
- [4] L. Kuczynski and K. Daly, “Qualitative methods for inductive (theory-generating),” *Handbook of dynamics in parent-child relations*, pp. 373–392, 2003.
- [5] P. Garg, C. Löding, P. Madhusudan, and D. Neider, “Ice: A robust framework for learning invariants,” in *In Proc. of 26th Int. Conf. on Computer Aided Verification*, 2014, pp. 69–87.
- [6] P. Garg, D. Neider, P. Madhusudan, and D. Roth, “Learning invariants using decision trees and implication counterexamples,” in *ACM Sigplan Notices*, vol. 51, no. 1. ACM New York, NY, USA, 2016, pp. 499–512.
- [7] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, “DistAI: Data-Driven automated invariant learning for distributed protocols,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021, pp. 405–421.
- [8] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 614–630.
- [9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” in *Science of computer programming*, vol. 69, no. 1-3. Elsevier, 2007, pp. 35–45.
- [10] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 291–301.
- [11] Y. Xia, D. Sur, A. S. Pingle, J. Deshmukh, M. Raghothaman, and S. Ravi, “Llm-guided predicate discovery and data augmentation for learning likely program invariants,” in *To appear in The 40th ACM/SI-GAPP Symposium On Applied Computing*, 2025.
- [12] G. J. Holzmann and W. S. Lieberman, *Design and validation of computer protocols*. Prentice hall Englewood Cliffs, 1991, vol. 512.
- [13] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The journal of logic and algebraic programming*, vol. 78, no. 5, pp. 293–303, 2009.
- [14] L. Gao, M. Lu, L. Li, and C. Pan, “A survey of software runtime monitoring,” in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2017, pp. 308–313.
- [15] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
- [16] G. J. Holzmann, “The model checker spin,” in *IEEE Transactions on software engineering*, vol. 23, no. 5. IEEE, 1997, pp. 279–295.
- [17] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [19] L. De Moura and N. Björner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [20] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, “I4: incremental inference of inductive invariants for verification of distributed protocols,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 370–384.
- [21] G. Byrd and M. Flynn, “Producer-consumer communication in distributed shared memory multiprocessors,” *Proceedings of the IEEE*, vol. 87, no. 3, pp. 456–466, 1999.
- [22] Z. Yang, M. Dai, and J. Guo, “Formal modeling and verification of smart contracts with spin,” *Electronics*, p. 3091, 2022.
- [23] C. Flanagan and K. R. M. Leino, “Houdini, an annotation assistant for esc/java,” in *International Symposium of Formal Methods Europe*, 2001, pp. 500–517.
- [24] P. Ezudheen, D. Neider, D. D’Souza, P. Garg, and P. Madhusudan, “Horn-ice learning for synthesizing invariants and contracts,” in *Proc. of the ACM on Programming Languages*, vol. 2, 2018, pp. 1–25.
- [25] T. Nguyen, K. Nguyen, and M. B. Dwyer, “Using symbolic states to infer numerical invariants,” *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3877–3899, 2021.
- [26] J. Yao, R. Tao, R. Gu, and J. Nieh, “Duoai: Fast, automated inference of inductive invariants for verifying distributed protocols,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 485–501.
- [27] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, “Can large language models reason about program invariants?” in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.
- [28] C. Janßen, C. Richter, and H. Wehrheim, “Can chatgpt support software verification?” in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2024, pp. 266–279.
- [29] 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,” *arXiv preprint arXiv:2310.09342*, 2023.
- [30] M. Valipour, B. You, M. Panju, and A. Ghodsi, “Symbolicgpt: A generative transformer model for symbolic regression,” *arXiv preprint arXiv:2106.14131*, 2021.
- [31] H. Wu, C. Barrett, and N. Narodytska, “Lemur: Integrating large language models in automated program verification,” *arXiv preprint arXiv:2310.04870*, 2023.
- [32] G. Wu, W. Cao, Y. Yao, H. Wei, T. Chen, and X. Ma, “Llm meets bounded model checking: Neuro-symbolic loop invariant inference,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 406–417.

Unlocking Hardware Verification with Oracle Guided Synthesis

Leiqi Ye¹, Yixuan Li¹, Guy Frankel¹, Jianyi Cheng¹, Elizabeth Polgreen¹

¹University of Edinburgh, UK

{leiqi.ye, yixuan.li.cs, G.Frankel-1, jianyi.cheng, elizabeth.polgreen}@ed.ac.uk



Abstract—Hardware verification is essential to ensure that hardware designs meet their design specifications and function as intended. However, use of formal verification requires extensive manual work in order to write formal specifications. Specification mining aims to alleviate this manual burden, with conventional techniques using statistical methods and pattern matching to generate likely specifications from sample execution traces from the hardware. A limitation of this is that the quality of the specifications is determined by the quality of the traces. In this paper, we present an approach that uses oracle-guided synthesis to generate specifications for hardware, using counterexamples and negative examples to refine specifications generated based on traces. We evaluate our approach on real-world Verilog benchmarks and demonstrate that specifications generated by our tool can detect high proportions of hardware mutations.

I. INTRODUCTION

Hardware verification ensures that a hardware design meets its specification(s) and functions correctly by identifying design flaws before manufacturing. Traditional hardware verification can either be done through simulation or formal verification. Simulation tests a design on a set of predefined test cases, potentially missing corner cases that may cause errors; whereas formal verification captures all possible cases under a set of specifications given by the designer. Formal verification is widely used in hardware verification today due to its better coverage, because a single undetected bug can necessitate a redesign and re-fabrication, incurring millions of dollars in losses.

However, a major challenge in hardware formal verification is writing formal specifications for a given hardware design. Manually writing specifications is time-consuming, usually taking up to 60%-70% of the development time [40], and requires significant expertise due to the complexity of both hardware design and its requirements [18], [39]. As a result, there is substantial interest in automatically generating specifications from arbitrary hardware designs. This process is usually referred to as specification mining [4], [31], [32], and has been applied to software and hardware. Most specification mining algorithms are passive and based on inferring specifications using pattern matching on traces passively collected from a system. These approaches rely on the set of traces being sufficiently comprehensive to capture the behavior of the model, and need carefully designed property templates to avoid overfitting to the traces.

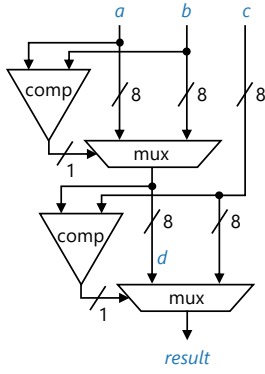
In this paper, we present Specification Mining of Assertions, Refined via Traces (SMART): a specification mining method for hardware designs based on oracle-guided inductive synthesis. Our synthesis process is based on 3 types of examples: we collect simulation traces from the hardware design, which we convert into *positive examples*, we use a model checker to generate unreachable states, which are treated as *negative examples*, and once the synthesizer has proposed a candidate assertion, we use a model checker to generate *counterexamples* if the assertion does not fully capture the behavior of the hardware.

We evaluate our approach on Verilog benchmarks from the literature and assess the specifications generated by our approach by mutating the hardware benchmark under test. A specification that can differentiate between the original design and a high number of mutant hardware designs is likely to be a *meaningful* specification. Assertions generated by SMART can differentiate a high proportion of mutants, achieving a 100% mutation detection rate on several real-world benchmarks. In comparison to related work, our approach infers meaningful specifications without requiring custom templates or carefully provided trace sets and provides formal guarantees that the specifications hold.

II. MOTIVATING EXAMPLE

Figure 1a shows a digital circuit that determines the maximum value among three 8-bit inputs. The circuit compares the values between the first two inputs *a* and *b* and passes the larger value for the next comparison, annotated as *d*. The result is then compared with the third input, where the maximal value is determined and sent to the output *result*. The corresponding source that generates the digital circuit is shown in Figure 1b, described in Verilog, a hardware description language (HDL) for digital circuits [43]. The Verilog source contains three main components: IO declaration, hardware description, and property assertion. First, both the three inputs and the output of the digital circuit are specified in lines 4-7 in eight bits. Second, lines 9-13 describe the design of the circuit in Figure 1a using two conditional assignments. Finally, lines 15-19 describe the invariants of the digital circuit design, also known as design specifications.

The first two parts form a design description that can be directly mapped to a digital circuit. Descriptions of real-world



(a) A digital circuit that determines the maximal value among three 8-bit inputs. comp = comparator, and mux = multiplexer.

```

1 // Return the max value
2 // of three inputs.
3 module max3 (
4     input  [7:0] a,
5     input  [7:0] b,
6     input  [7:0] c,
7     output [7:0] result
8 );
9     wire [7:0] d;
10    // d = max(a, b)
11    assign d = (a>=b) ? a:b;
12    // result = max(d, c)
13    assign result = (d>=c) ?
14        d:c;
15
16    // assertions
17    assert property
18        (a <= result);
19    assert property
20        (d <= result);
21 endmodule

```

(b) The Verilog and SVA source for the digital circuit.

Fig. 1: A motivating example. Existing work [21] only supports boolean circuits and cannot automatically generate SVAs for the example above. SMART automatically generates efficient SVAs for formal verification.

hardware designs are often more complex and finer-grained, down to the bit level, leading to a high risk of introducing design bugs. For example, the “>” character at line 13 may be mistakenly typed as “<”, leading to a syntactically correct but functionally incorrect condition $a \leq b$. Such design bugs could be captured by hardware verification using design invariants. The invariants for Verilog designs are described in a property specification language named SystemVerilog Assertions (SVA) [44], as defined in Section III-A. In verification, the circuit behavior is formally verified to match each SVA under any input values. The invariants on lines 15-19 of Figure 1b would fail if the circuit contained the previously described bug.

Still, existing approaches in hardware verification for a given Verilog design require human effort. A key challenge in automatically generating such specifications is the unique semantics of hardware description languages like Verilog compared to traditional software programs. Traditional static analysis techniques on software programs cannot be directly applied to digital circuit descriptions. For example, re-ordering statements in Verilog, such as swapping lines 11 and 13 in Figure 1b, does not affect the correctness of the output, while this usually causes errors in software programs that follow sequential execution. The analysis process could become more challenging when the digital design is implemented with complex obfuscation for security reasons [8].

Existing work uses customized static analysis to infer SVA [21]. However, it focuses on a small set of Verilog designs and only works under stringent constraints. For example, the hardware design must be described as a boolean circuit, while modern hardware designs contain wires of multiple bits. We seek a general approach for automated SVA generation.

This example also illustrates a key limitation of template-based assertion mining tools: they often rely on fixed syntactic forms or single-variable patterns. In contrast, the assertions shown in Figure 1(b), such as assert property $(a \leq \text{result})$; and assert property $(d \leq \text{result})$;, involve multiple variables and relational reasoning across signal dependencies. These kinds of compound assertions fall outside the expressiveness of common pattern-based templates. SMART, by framing assertion generation as a formal synthesis problem, is able to automatically infer such expressive multi-variable properties without relying on pre-defined templates.

The SVAs should precisely describe the hardware behaviors with minimal tolerance for design discrepancies. We formalize the hardware verification problem into a formal synthesis problem and leverage existing program synthesis tools for the automated generation of efficient SVAs. In the rest of the paper, we will describe our approach in detail.

III. BACKGROUND

A. Verilog and SystemVerilog Assertions (SVAs)

Verilog is an HDL used to describe digital electronic systems. It has been widely used in digital system design and verification at register-transfer level (RTL). Compared to traditional software programming languages, such as C or Python, which describe the sequential execution of instructions, Verilog is tailored to describe parallel hardware behavior.

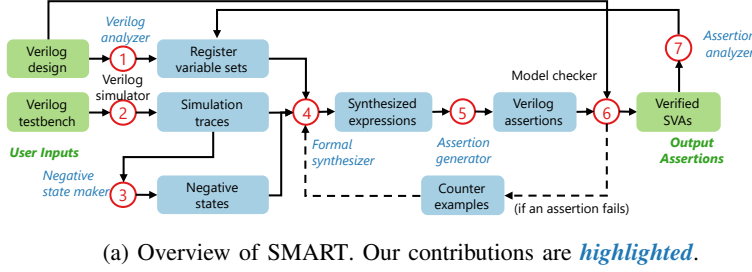
Most HDLs, including Verilog, provide two main programming styles for describing hardware designs: structural and behavioral descriptions. A structural description presents a circuit at a low level, specifying how individual components, such as gates, multiplexers, or registers, are interconnected. A behavioral description is used for high-level descriptions of hardware behaviors, focusing on what the circuit should do rather than how it is implemented. To maximize generality and avoid targeting particular use cases, our work supports both structural and behavioral Verilog inputs.

On the verification side, SVAs are used as an extension of the Verilog language for formal verification. Unlike traditional run-time assertions that are evaluated during simulation and are limited to runtime checking, SVAs specify properties that must always hold true, and can be checked with formal verification, to ensure that the implemented design meets its design specifications. In this work, we take Verilog and its SVA generation, as the example, to show how program synthesis can guide automated hardware verification. Our method is general and can be applied to other HDLs, for example VHDL [5] and its own Property Specification Logic [13].

B. Syntax-Guided Synthesis (SyGuS)

SyGuS is a formal framework for program synthesis, which automatically generates programs that satisfy a given specification. We frame the problem of generating assertions as a SyGuS problem.

A SyGuS problem has a closed formula of the form $\exists F. \forall \vec{x}. \phi$, where F is a function to be synthesized, $\vec{x}$ is a list of universally quantified variables, and ϕ is a first-order



```

1 (set-logic BV)
2 (synth-fun inv ((d (_ BitVec 8))
3 (result (_ BitVec 8))) Bool)
4 ; Constraints from the simulation
   trace:
5 (constraint (= (inv #142 #148) true)
6 (constraint (= (inv #172 #179) true)
7 )
8 ; Constraints from negative states:
9 (constraint (= (inv #111 #2) false))
   (check-synth)

```

(b) A simplified example of a synthesis conjecture generated from simulation traces and negative states. It reasons about the variables d and $result$ in the circuit in Figure 1.

Fig. 2: The SMART tool workflow and SyGuS example for invariant synthesis.

formula in a background theory T which specifies the semantic correctness constraints for F . The synthesis function F has a corresponding grammar, G . A valid solution to a SyGuS formula is a function f such that $\forall x. \phi[F \mapsto f]$ is T -satisfiable and $f \in \mathcal{L}(G)$, where $\mathcal{L}(G)$ denotes the language defined by G .

C. Oracle Guided Inductive Synthesis

SyGuS problems are solved by Oracle-Guided Inductive Synthesis algorithms (OGIS) [25]. Given a SyGuS problem as described above, OGIS alternates between a learner, which guesses or enumerates through candidate programs in G , and an oracle that the learner may query to check if the candidate program satisfies the specification, or to obtain feedback if it does not. The most common variant of OGIS is CounterExample Guided Inductive Synthesis (CEGIS) [41], where the learner proposes a candidate program f^* to the oracle. If the candidate program fails to satisfy the specification, i.e., $\exists x. \neg \phi[F \mapsto f^*]$, the oracle returns a *counterexample*, which is an assignment to x [41]. The learner stores a set of counterexamples C , and then searches for a new candidate function such that $\forall x \in C. \phi[F \mapsto f]$ is T -satisfiable.

However, other oracles, which provide other forms of guidance, are sometimes used. Prior work in invariant synthesis [16] uses oracles that provide positive examples (on which the invariant should evaluate to true) and negative examples (on which the invariant should evaluate to false) to guide the search.

IV. OVERVIEW

A. Problem Statement

Given a hardware design, we wish to find a set of assertions that are valid (i.e., never violated by the hardware design). We frame this as a formal synthesis problem. First, let us define what we mean by hardware design.

Definition 1: A hardware design H is a tuple $\langle V_H, I, S, s_0, T \rangle$

- where V_H is a set of m state variables.
- $V_i \subseteq V_H$ is a set of input variables, i.e., variables that can be assigned to by an external source.

- S is a set of states, where each state corresponds to a full assignment to all variables in V_H , i.e., $s \in S$ and $s = \{v_1 \mapsto c_1, \dots, v_m \mapsto c_m\}$ where $V_H = \{v_1, \dots, v_m\}$ and $c_1, \dots, c_m$ are constant literals.
- $I \subseteq S$ is a set of initial states.
- $T \subseteq S \times S$ is a predicate that defines the set of possible transitions between states.

A valid assertion or specification is a predicate over the state variables that is true for all reachable states, i.e., it is never violated by any execution of H . We define this formally as follows:

Definition 2: A sequence π of states $s_0, \dots, s_n$ is an execution in H if, for each $0 \leq i \leq n$, $\langle s_i, s_{i+1} \rangle \in T$. A state $s \in S$ is reachable if there is a trace $s_0, \dots, s$, where $s_0 \in I$, i.e., if there is a path from an initial state to s .

Definition 3: An assertion $\alpha \subseteq S$ is a predicate over (a subset of) the variables V_H . We use $\alpha(s)$ to indicate the result of evaluating the predicate α over the assignment to the state variables given by s . An assertion thus defines a set of states $S_\alpha = \{s \mid \alpha(s) = \top\}$. An assertion α *holds* if and only if $S_\alpha \subseteq R$, where R is the set of all reachable states. If α holds, we say that $H \models \alpha$.

We note that the generated specifications should also be *meaningful*: they should not be vacuously true, and they should be compact and more readable than the hardware model itself. We evaluate our specifications against these criteria using mutation detection, described in Section VII-A2.

a) Formal Problem Statement: Formally, given a hardware design H (see Definition 1), we wish to generate a set of assertions $\alpha_1, \dots, \alpha_n$ such that $H \models (\alpha_1 \wedge \dots \wedge \alpha_n)$. Thus, the synthesis conjecture we wish to solve is $\exists \alpha_1, \dots, \alpha_n. \forall V_H. \phi$. The terms are defined as follows.

- $\alpha_1 \dots \alpha_n$ are the assertions to be synthesized.
- V_H is the set of variables in H .
- ϕ is the specification, which should state that $H \models (\alpha_1 \wedge \dots \wedge \alpha_n)$.

We also provide a grammar for $\alpha_1, \dots, \alpha_n$, which serves to reduce overfitting as well as reduce the search space of possible assertions.

B. Proposed SMART Framework

We will explain how we use synthesis from positive and negative examples to overcome the above challenges. Specifically, Section V describes how we collect simulation traces from H and use these to construct the grammar, and Section VI describes the construction of our oracle-guided synthesizer. An overview of the SMART framework is illustrated in Figure 2a:

- ① A lightweight static analyzer parses the Verilog design to identify the key variables (register variable sets in the figure) used to construct grammars for the syntax-guided synthesis, described in Section V-B;
- ② For given user inputs, a hardware simulator generates a set of simulation traces as *positive examples*, described in Section V-A;
- ③-⑤ A random search is called with a hardware model checker to find unreachable states in the hardware design, known as *negative examples*, which are used to construct a synthesis conjecture, shown in Figure 2b together with the simulation traces and variables for an oracle-guided program synthesizer, leading to synthesized assertions that satisfy the synthesis conjecture and are later translated to syntactically correct SVAs, described in Section VI;
- ⑥ The assertions are checked against the hardware using an unbound model checker, resulting in either verification success (passing assertions to the output) or failure with *counterexamples* (refining the synthesis conjecture to produce more assertions), described in Section VI-C;
- ⑦ The verified assertions are analyzed to refine the variable set used to generate the grammar, accelerating the mining process, described in Section VI-D.

V. TRACE COLLECTION & GRAMMAR CONSTRUCTION

The core idea behind SMART is to use formal synthesis to generate specifications for hardware designs. The inputs to our formal synthesis are a set of variables to be used in the specification, as well as a set of positive examples, negative examples, and counterexamples. In this section, we describe how we obtain the variable set, the positive examples, and the negative examples.

A. Verilog Simulation

Here, we explain how we obtain positive examples through simulation traces obtained from the Verilog simulator. We assume that a hardware design H comes with a corresponding hardware testbench. A hardware testbench is a simulation environment, used with a simulator to test the states of a hardware design by simulation. Different values are fed into the inputs V_i of the hardware design at different time steps, and the outputs are checked to detect potential discrepancies compared to given expectations. This leads to a simulation trace for each test run. A simulation trace contains a subset of internal wire variables V_H and records their states over a sequence of time stamps. For example, a trace may include an event where both `a` and `result` increase from 8 to 10 while

`b` and `c` are both 5 at the time stamp 50 nanoseconds after the start of the simulation.

Each simulation trace is translated into an execution trace (see Definition 2).

B. Identifying the Variable Sets: Verilog Analysis

Formal synthesis in general relies on enumerating the space of programs, leading to a space growing exponentially with the number of variables. To reduce this problem, SMART uses heuristics to select only random subsets of variables for assertion synthesis. We use a lightweight analysis of the simulation trace and Verilog code to extract the register variables that only change values with clock edges. We experimented with heuristic-based selection strategies – variable dependency analysis, which attempts to select semantically related variables. However, empirical evaluation showed the method provides limited benefit in our setting.

Specifically, we identify the register variables from the hardware, denoted V_H . For a given trace π , we define a corresponding set of trace variables V_π to be the set of variables that change at some point during the trace. We use $s_t(v_i)$ to indicate the value assigned to variable v_i at time step s_t in one trace. That is $V_\pi = \{v \in V_H \mid \exists s_i, s_{i+1} \in \pi \wedge s_i(v) \neq s_{i+1}(v)\}$. Given a collection Π of traces, let $V_\Pi = \bigcup_{\pi \in \Pi} V_\pi$ be the union of all trace variables appearing in Π . This ensures we consider only variables that feature in the trace and are also in the Verilog design, i.e., variables introduced by the simulator and parameters defined but never used in the design are never considered.

SMART adopts a randomized variable subset selection strategy. We randomly sample a small subset of variables $V_i \subset V_\Pi$. These variable sets are updated at each iteration of SMART. Suppose α is synthesized from a subset $V_i \in \mathcal{V}$; Once α is produced, the assertion analyzer updates $V_i \leftarrow V_i \setminus \text{var}(\alpha)$, where $\text{var}(\alpha)$ returns the variables used in α . This avoids reusing those same variables in future assertions. SMART continues this process, reducing the size of the variable set V_i in each iteration until either $V_i = \emptyset$, or the synthesis problem is infeasible with the remaining variables, or a generated assertion is discovered to be invalid via formal verification. Once one of these conditions is satisfied, SMART moves on to considering V_{i+1} . The size of the initial subset is determined by the total number of variables in V_Π , that is:

$$|V_i| = \begin{cases} |V_\Pi| - 1, & \text{if } |V_\Pi| \leq 5 \\ 5, & \text{if } 5 < |V_\Pi| \leq 400, \\ 20, & \text{if } 400 < |V_\Pi|, \end{cases}$$

where $|V|$ indicates the number of variables in the set V . This ensures the extracted variables efficiently capture key hardware states, leading to more meaningful assertions.

VI. ORACLE-GUIDED SYNTHESIZER

In this section, we describe our Oracle Guided Inductive Synthesis approach to generating SVA assertions, as shown in Algorithm 1.

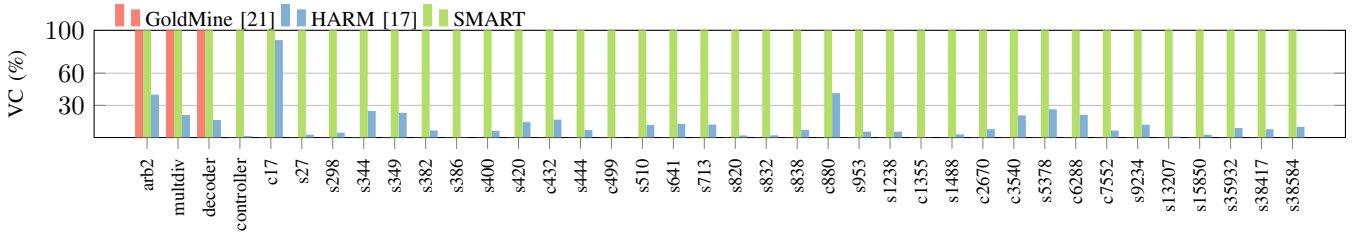


Fig. 3: Verification Correctness (VC) rates of different approaches over various benchmarks.



Fig. 4: Mutation detection (MD) rates of different approaches over various benchmarks.



Fig. 5: Impact of Counterexamples on SMART's Mutation Detection Performance.

A. Constructing the Grammar G

The first step of the synthesis is to generate a grammar for the synthesis problem. Given a variable set V and a background theory, SMART constructs the following grammar:

$$\begin{aligned}
\text{Bool} ::= & \text{Bool} \simeq \text{Bool} \mid \neg \text{Bool} \mid \text{Bool} \wedge \text{Bool} \mid \\
& \text{Bool} \vee \text{Bool} \mid \text{any boolean } v \text{ in } V \\
Bv ::= & Bv \simeq Bv \mid Bv \geq Bv \mid Bv \leq Bv \mid \\
& Bv < Bv \mid Bv > Bv \mid Bv \geq_u Bv \mid \\
& Bv \leq_u Bv \mid Bv <_u Bv \mid Bv >_u Bv \\
Bv ::= & -Bv \mid \sim Bv \mid Bv \& Bv \mid Bv \circ Bv \mid Bv + Bv \mid \\
& Bv - Bv \mid Bv * Bv \mid \text{any bitvector } v \text{ in } V
\end{aligned}$$

$\circ$ denotes bitwise OR, and $\&$ denotes bitwise AND operation. X_u is the unsigned version of the comparator X , e.g., $<_u$ indicating an unsigned $<$. $\simeq$ denotes equality, and $=$ denotes assignment. SMART omits lines marked $*$ if there are no bitvector variables in V .

B. Positive and Negative Examples

Recall that our specification ϕ for our synthesis problem for synthesizing $A = \alpha_1 \dots \alpha_n$ is that $H \models A$. It is infeasible to give this specification to an off-the-shelf SyGuS solver, which implements CEGIS, because encoding the formula $H \models A$ requires unwinding the transition system of H and suffers from the classic model-checking state-space explosion problem [11]. Instead, we construct an OGIS loop based on using positive and negative examples, combined with counterexamples.

A *positive example* s^+ is a reachable state in H . The hardware $H \models \alpha$ if and only if $\alpha(s^+) = \top$.

A *negative example* s^- is an unreachable state in H . We use negative examples to refine the assertions, and thus we require that $\alpha(s^-) = \perp$.

We generate a set of positive examples, $\mathcal{P}$ from the simulation traces. For every step in a trace $\pi = s_0, s_1 \dots$, we generate a single positive example. In order to generate negative examples, we randomly generate assignments to the state variables. This gives us a random state s . We then check whether $s \in R$ using a model checker with k-induction. If s is not reachable, we add it to a set of negative examples $\mathcal{N}$.

Algorithm 1 Oracle Guided Inductive Synthesis

```
1: procedure OGIS( $H, \mathcal{P}, \mathcal{N}, \mathcal{V}$ )
2:    $A \leftarrow \emptyset, i \leftarrow 0$ 
3:    $V \leftarrow \mathcal{V}, G \leftarrow \text{GENERATEGRAMMAR}(V)$ 
4:   while true do
5:      $\alpha \leftarrow \text{ENUMERATE}(G, \mathcal{P}, \mathcal{N})$ 
6:     if  $\text{VERIFY}(\alpha, H)$  then
7:        $A \leftarrow A \cup \alpha$ 
8:        $V \leftarrow V \setminus \text{var}(\alpha)$ 
9:       if  $V = \emptyset$  then
10:         $i \leftarrow i + 1$ 
11:         $V \leftarrow \mathcal{V}$ 
12:         $G \leftarrow \text{GENERATEGRAMMAR}(V)$ 
13:       end if
14:     else
15:        $c \leftarrow \text{VERIFY.GET\_CEX}$ 
16:        $\mathcal{P} \leftarrow \mathcal{P} \cup \{c\}$ 
17:        $i \leftarrow i + 1$ 
18:       if  $i = |\mathcal{V}|$  then return  $A$  ▷ No more variable sets
19:       end if
20:        $V \leftarrow \mathcal{V}$ 
21:        $G \leftarrow \text{GENERATEGRAMMAR}(V)$ 
22:     end if
23:   end while
24: end procedure
```

Given a set of positive and negative examples, SMART uses program synthesis to synthesize an assertion that evaluates to true for the positive examples and false for the negative examples. That is, it solves the synthesis conjecture $\exists \alpha (\forall s \in \mathcal{P}. \alpha(s) = \top \wedge \forall s \in \mathcal{N}. \alpha(s) = \perp)$. We use an off-the-shelf enumerator built into cvc5 [6] to determine α . Given α , SMART applies simple syntactic translation rules to translate the assertions to syntactically correct SVAs for verification.

In our implementation, we generate one negative example per iteration. This is based on our observation that a single unreachable state is often sufficient to eliminate trivial or vacuously true assertions during synthesis. Using only one example keeps the synthesis problem small and improves solver performance. This design choice strikes a balance between counterexample strength and synthesis tractability, and we found in practice that increasing the number of negative examples per iteration did not significantly improve assertion quality or mutation detection rate.

C. Counterexamples

The generated SVAs are verified against H using the same model checker as the negative example generation, using k-induction [38]. We use $H \not\models \alpha$ to indicate that the hardware model H violated the assertion α , and $H \models \alpha$ to indicate that H did not violate the assertion.

If $H \not\models \alpha$, the verifier provides a counterexample trace $\pi_{\text{cex}} = s_0 s_1 \dots s_k$ where $\alpha(s_k) = \perp$. All states in π_{cex} are, by definition, reachable, and so an α such that $H \models \alpha$ should evaluate to true on any state in π_{cex} . SMART adds all states in the counterexample to the set of positive examples.

In practice, counterexamples are frequently generated during the synthesis process, especially in early iterations where candidate assertions are often overly general. These counterexamples play a critical role in refining the synthesized

assertions by eliminating vacuously true or weak predicates. We observe that incorporating counterexamples significantly improves both the quality and effectiveness of the final assertion set. The impact of counterexample refinement is further discussed in the section VII.

D. Assertion Analysis

If α is valid, SMART adds it to the set of assertions A , removes the variables in the assertion from the current variable set, and repeats the synthesis process. If the variable set is then empty, SMART moves on to the next variable set. The assertion analyzer also removes any syntactic duplicates. The iterative process terminates either when $|A|$ reaches a pre-defined limit, or we reach a pre-defined time-out, or all variable sets are exhausted.

VII. EVALUATION

We implement SMART, using cvc5 [6] version 1.2.0 to enumerate assertions, and EBMC [1] version 5.5 to verify the correctness of the generated SVAs and as the model checker that provides counterexamples inside the synthesis process. We use Cocotb [37] as the Verilog simulator. We use benchmarks from the related work [21] and two standard low-level hardware benchmark sets, ISCAS'85 and ISCAS'89 [2]. The benchmark set published by GoldMine includes a set of realistic digital circuit designs in behavioral Verilog: *arb2*, *controller*, *decoder*, *ld_stage* and *multdiv*, taken from the existing Ibex RISC-V CPU implementation [36]. There are also two standard benchmarks to describe low-level digital circuit designs. The ISCAS'85 benchmark [19] contains a set of combinatorial circuits, which consist of logic gates only; and the ISCAS'89 benchmark [7] which contains a set of sequential circuits whose hardware behaviors depend on clock signals. We evaluated SMART, HARM [17] and GoldMine [21] over all the benchmarks that have available Verilog sources. Result format converters are added into the workflow of HARM and GoldMine to make their result into SVA to run our evaluator. This is provided in the artifacts.

A. Evaluation Metrics

We evaluate the quality of the generated specifications by SMART based on the following two metrics:

1. **Correctness:** the inferred specifications should correctly describe the hardware behavior;
2. **Meaningfulness:** the specifications should differentiate between the original design and a high number of mutant hardware designs.

These are described in detail in the following sections.

1) *Correctness of Assertion Sets:* All assertions generated by SMART are correct with respect to the hardware design, as verified by the model checker. As not all the tools provided in related works use formal verification, some assertions generated by these works do not hold on the original model, i.e., $H \not\models A$.

a) *Formal Verification of SVAs*: We verify all assertions generated by the related work using k-induction. For each assertion, we run it on the original hardware model; if it passes verification, we keep it; otherwise, we discard it. We then compute the verified correctness rate, VC-rate, which measures the proportion of assertions that correctly reflect the model. Suppose that A_{gen} is the set of generated assertions for one hardware design, and $A_{ver} = \{a \in A_{gen} \mid H \models a\}$ is the set of assertions that pass verification. We calculate the VC-rate as $\frac{|A_{ver}|}{|A_{gen}|}$.

2) *Meaningfulness of Assertion Sets*: In order to assess how meaningful our specifications are, we use mutation tests, as is standard in the literature [23]. Intuitively, a good specification should be one that can detect if we change the underlying hardware design. To enable a fair comparison with other tools when performing mutation testing, we only perform testing on the assertions that pass verification.

a) *Mutant Generation*: We use the mutation rules from Repinski *et al.* [22], as shown in Figure 6. The left-hand side indicates the operators to which a rule can be applied, and the right-hand side indicates the result, for example, rule r_{g1} replaces any gate operator with $\vee$. In general, rules $r_{g1} \dots r_{g8}$ are applicable to gate operators. Rules $r_{b1} \dots r_{b3}$ are applicable to bitwise operators. Rules $r_{a1}, \dots, r_{a5}$ are applicable to arithmetic operators. Rules $r_{r1} \dots r_{r6}$ are applicable to relational operators, and we use $\simeq$ to indicate equality. The rule r_{neg} negates any boolean variable, and r_{ran} assigns a random value to any bitvector literal.

We iterate over the Verilog file, and for each operator, variable, and literal we encounter, we identify the set of applicable rules. We then randomly choose one rule from this set, and apply it to generate a new mutant. This generates a randomly chosen subset of all possible mutations that can be generated by applying one mutation rule to H . This is shown in Algorithm 2.

b) *Mutant Detection*: Given a hardware design H and a conjunction of synthesized assertions A , we mutate H using a pre-defined set of rewrite rules to generate a set of hardware designs $\mathcal{H}_{mut}$. A mutant H' can be detected by our set of assertions A if $H \models A$ and $H' \not\models A$. We use k-induction, as implemented in EBMC, to determine whether a mutant can be detected by a given assertion set.

We define the mutation detection rate to be:

$$\text{MD-rate} = \frac{|\{H' \in \mathcal{H} \mid H' \not\models \alpha\}|}{|\mathcal{H}|}$$

B. Baselines

We compare SMART to two related works: HARM [17] and GoldMine [21]. HARM is based on learning patterns from simulation traces, and we run it with 10,000-cycle random input traces. GoldMine combines static analysis with decision-tree-based data mining for SVA generation and uses the closed-source JasperGold tool to verify the correctness of the generated assertions. We replace JasperGold with EBMC as the formal verification tool, and the raw output from

$(\wedge \mid \uparrow \mid \downarrow \mid \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g1}} \vee$	$(- \mid * \mid / \mid \%) \xrightarrow{r_{a1}} +$
$(\vee \mid \uparrow \mid \downarrow \mid \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g2}} \wedge$	$(+ \mid * \mid / \mid \%) \xrightarrow{r_{a2}} -$
$(\vee \mid \wedge \mid \downarrow \mid \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g3}} \uparrow$	$(+ \mid - \mid / \mid \%) \xrightarrow{r_{a3}} *$
$(\vee \mid \wedge \mid \uparrow \mid \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g4}} \downarrow$	$(+ \mid - \mid * \mid \%) \xrightarrow{r_{a4}} /$
$(\vee \mid \wedge \mid \uparrow \mid \downarrow \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g5}} \oplus$	$(+ \mid - \mid * \mid /) \xrightarrow{r_{a5}} \%$
$(\vee \mid \wedge \mid \uparrow \mid \downarrow \mid \oplus \mid \neg \mid buf) \xrightarrow{r_{g6}} \odot$	$(\neq \mid > \mid < \mid \geq \mid \leq) \xrightarrow{r_{r1}} \simeq$
$(\vee \mid \wedge \mid \uparrow \mid \downarrow \mid \oplus \mid \odot \mid buf) \xrightarrow{r_{g7}} \neg$	$(\simeq \mid > \mid < \mid \geq \mid \leq) \xrightarrow{r_{r2}} \neq$
$(\vee \mid \wedge \mid \uparrow \mid \downarrow \mid \oplus \mid \odot \mid \neg) \xrightarrow{r_{g8}} buf$	$(\simeq \mid \neq \mid < \mid \geq \mid \leq) \xrightarrow{r_{r3}} >$
$(\odot \mid \wedge) \xrightarrow{r_{b1}} \&$	$(\simeq \mid \neq \mid > \mid \geq \mid \leq) \xrightarrow{r_{r4}} <$
$(\& \mid \wedge) \xrightarrow{r_{b2}} \circ$	$(\simeq \mid \neq \mid > \mid < \mid \leq) \xrightarrow{r_{r5}} \geq$
$(\& \mid \odot) \xrightarrow{r_{b3}} \wedge$	$(\simeq \mid \neq \mid > \mid < \mid \geq) \xrightarrow{r_{r6}} \leq$
bitvector literal $\xrightarrow{r_{ran}}$ random value	bool $\xrightarrow{r_{neg}} \neg \text{bool}$

Fig. 6: Mutation rules. We use $\uparrow, \downarrow, \oplus, \odot$ to represent NAND, NOR, XOR, and XNOR respectively. $\&, \circ$, and $\wedge$ represent bitwise AND, OR and XOR respectively. *buf* indicates the Verilog operator *buf*, which transfers an input to an output without changing polarity. This is thus equivalent to removing an operator.

Algorithm 2 Generation of Hardware Mutants

```

1: procedure GENERATEMUTANTS(Verilog file  $F$ , Rules  $R$ )
2:    $Mutants \leftarrow \emptyset$ 
3:   for  $op$  in  $F$  do ▷ Iterate over operators in file
4:      $R_{app} \leftarrow \text{FINDAPPLICABLERULES}(R, op)$ 
5:      $r \leftarrow \text{RANDOMSAMPLE}(R)$ 
6:      $F' \leftarrow \text{MUTATE}(F, op, r)$ 
7:      $Mutants \leftarrow F'$ 
8:   end for
9:   for boolean var  $v$  in  $F$  do
10:     $F' \leftarrow \text{MUTATE}(F, op, r_{neg})$ 
11:     $Mutants \leftarrow F'$ 
12:   end for
13:   for literals  $l$  in  $F$  do
14:     $F' \leftarrow \text{MUTATENUMERAL}(F, op)$ 
15:     $Mutants \leftarrow F'$ 
16:   end for
17:   return  $Mutants$ 
18: end procedure

```

GoldMine is included in the benchmark for comparison. Unfortunately, GoldMine only supports behavioral Verilog. Both tools generate assertions in their own formats. For HARM, we provide a translation tool to convert its output into SVA. For GoldMine, we manually process its raw data into SVA. The resulting SVAs from both tools are evaluated using the metrics mentioned above.

C. Results

Table I shows a summary of the evaluation results.

a) *VC-rate*: We run formal verification of the generated assertions by various approaches and report the VC-rates in Figure 3. Overall, most of the assertions generated by HARM failed the verification in EBMC with returned counterexamples. This is because HARM infers the specifications of de-

TABLE I: Summary of results comparing SMART and related work. We report the average size of the generated assertion set A_{gen} , VC-rate and MD-rate across all benchmarks. The # column reports the number of benchmarks supported by each approach.

	Structural Verilog (34 benchmarks)				Behavioral Verilog (4 benchmarks)				All (38 benchmarks)			
	#	$ A_{gen} $	VC-rate	MD-rate	#	$ A_{gen} $	VC-rate	MD-rate	#	$ A_{gen} $	VC-rate	MD-rate
GoldMine	0	0	n/a	0.0%	4	38.8	100%	21.16%	4	38.8	100%	4.3%
HARM	34	60676	8.41%	21.39%	4	3655	8.6%	29.6%	38	54674	8.4%	22.25%
SMART	34	1069	100%	55.41%	4	47	100%	25.17%	38	943.5	100%	52.23%

signs purely from traces and validates them using simulations, which could miss corner cases. The low rate persists even with long traces: in our experiments, HARM uses traces with a length of 10,000 cycles, which is significantly longer than the traces used for the other tools.

Both GoldMine and SMART always generate verified assertions for benchmarks, leading to 100% VC rates, but GoldMine only supports a fraction of the benchmarks (the behavioral Verilog benchmarks).

b) MD-rate: The mutation detection results are summarized in Figure 4. Overall, SMART achieves a higher MD-rate than any related work on 32/38 benchmarks. In the 4 behavioral Verilog benchmarks supported by GoldMine, SMART produces comparable results, with both achieving an average MD-rate around 25% using a comparable number of assertions. In comparison to HARM, across the whole benchmark set, SMART achieves higher MD-rates on 33 of the benchmarks, with an average MD-rate 2.62x higher than HARM.

The MD-rates achieved by SMART vary significantly across benchmarks, reflecting the complexity and structural differences of the circuits. For small, medium, and large benchmarks, we achieve a strong assertion-based detection rate. However, in the case of extremely large circuits, timeouts limit the number of generated assertions, leading to a lower mutation detection rate.

Some benchmarks do not achieve a high mutation detection rate. For SMART, this may be because these benchmarks require assertions that reason about multiple time steps to differentiate between some mutations. However, it is worth noting that some mutations may not change the behavior of the underlying circuit, so a mutation detection rate of 100% may not be possible on all benchmarks.

D. Readability

The readability of specifications is subjective. Here, we discuss quantitative metrics that we use to approximate how readable the assertion sets are.

First, we look at the size of the assertion sets produced. The average assertion set size produced by SMART is $> 50\times$ smaller than the assertion sets produced by HARM. The assertion sets produced by GoldMine are of comparable size to SMART, on the limited supported benchmarks.

Second, we implement an approximate equivalence checker to estimate the amount of redundancy in the assertion sets.

TABLE II: Examples of boolean rewrite rules used in for equivalence rewriting. The rewriting system consisted of 105 rules for bitvectors and 10 for booleans.

Pattern	Transformation
$a = \neg a$	$\perp$
$a = b$	$b = a$
$a \& b$	$b \& a$
$a \& (b \& c)$	$(a \& b) \& c$
$a \parallel b$	$b \parallel a$
$a \parallel (b \parallel c)$	$(a \parallel b) \parallel c$

We use an approximation of equivalence as a complete equivalence check of all assertions is intractable. Specifically, we use a non-destructive rewrite system based on equivalence graphs [45] to rewrite the set of assertions returned by SMART and check whether any assertions are equivalent. An example of the rewrite rules used is in Appendix II. We find that, in all cases, the rewrite system reduces the size of the assertion sets produced by SMART by less than 3%, suggesting SMART assertion sets contain relatively little redundancy.

E. Rewrite system

The rewriting system used to evaluate assertion redundancy was done using an equivalence graph (e-graph). E-graphs provide a constructive method to apply rewrite rules to capture semantically equivalent terms. We use `egg`[45] to generate an e-graph for the assertion sets of each benchmark. We then test whether equivalences exist in the assertion sets, using rewrite rules from CVC5 [6], Bitwuzla [33], and E-syn [9].

VIII. EVALUATION DETAILS

A. Behavioral vs. Structural Verilog

We observe that SMART achieves comparable verification correctness (VC) rates on both behavioral and structural Verilog benchmarks, but mutation detection (MD) rates are generally higher for structural designs. This difference can be attributed to three main factors:

- 1) **Control/Data Path Complexity.** Behavioral Verilog often encapsulates complex control and data path logic using higher-level constructs (e.g., `if`, `case`, `always @(*)`), making the synthesized assertions less precise in targeting low-level signal interactions. In contrast, structural Verilog exposes finer-grained gates and flip-flops, enabling SMART to more effectively synthesize assertions that tightly capture actual hardware behavior.

- 2) **Grammar and Variable Set Size.** Structural designs tend to contain more low-level signals (e.g., internal nets, intermediate registers), resulting in larger variable sets and more expressive grammars. While this increases synthesis cost, it also gives the synthesizer more flexibility to discover strong invariants. In behavioral designs, the variable space is often smaller and more abstract, limiting synthesis diversity.
- 3) **Signal Variability.** We found that structural Verilog exhibits more frequent value changes in simulation traces due to its gate-level representation, leading to richer positive example sets. This makes it easier for SMART to distinguish between correct and mutated designs. Behavioral designs, by contrast, often abstract away low-level toggling, which may result in sparser and less informative traces.

These differences explain why SMART achieves stronger mutation detection on structural designs (average MD-rate 55.4%) compared to behavioral ones (25.17%)

B. Effect of Counterexample Refinement

We conducted an ablation study to quantify the effect of counterexample-based refinement in SMART. Specifically, we disabled the counterexample refinement mechanism described in Section VI-C, and re-ran the synthesis process using the same set of benchmarks and initial parameters.

The results are summarized in Figure 5. Across all benchmarks, removing refinement led to a consistent drop in mutation detection (MD) rate. The average MD-rate dropped from 63.1% to 32.1%, a 49% relative reduction. In some cases (e.g., *arb2*, *c1355*, *s13207*), MD-rate dropped to zero, indicating that the initial assertion was vacuously true and not filtered out without refinement. Only a few small circuits (e.g., *s27*, *s349*) showed minimal sensitivity to refinement.

These results confirm that counterexample refinement plays a critical role in eliminating weak or vacuous assertions. Without it, SMART often converges on trivial assertions that pass simulation but fail to distinguish between correct and mutated designs.

C. Limitations

Currently, SMART generates assertions that reason about a single time step, and is unable to capture behaviors that evolve over multiple cycles. Extending its synthesis capability to temporal properties would be feasible, but would increase the size of the variable sets that the synthesis procedure needs to consider. Specifically, we would need to introduce a copy of the original variable set for each time step we wish to consider in the temporal property, so a property reasoning over two time steps would duplicate the variable set. This is why related work requires the user to provide restrictive templates, which limit the temporal operators used in temporal logic properties. SMART may be complementary to this by providing predicates that can be used within such templates.

The runtime of SMART is limited by verification time. One possible optimization for SMART would be to initially use

testing to discard assertions that can easily be falsified with testing. However, ultimately, verification is a necessary part of the SMART pipeline in order to allow us to generate *correct* assertions.

IX. RELATED WORK

Automated assertion generation is a critical area of research for improving hardware verification. Developing static program analysis for SVA generation is actively being studied. GoldMine [21] combines static analysis and decision-tree-based data mining for SVA generation, supporting boolean circuits.

GoldMine’s inability to support structural Verilog stems from its reliance on semantic analysis tailored to behavioral Verilog. The analysis framework assumes high-level constructs and control flow information typically found in behavioral code. However, structural Verilog presents a flat netlist-like representation, where connectivity is expressed via gate and wire instances rather than procedural code. Supporting structural Verilog would therefore require fundamentally different semantic analysis techniques, such as netlist-level reasoning, instance graph traversal, and low-level dependency tracking—none of which are addressed in GoldMine. As a result, extending GoldMine to handle structural designs is non-trivial and would require substantial redesign of its core algorithms. By contrast, SMART supports both behavioral and structural Verilog inputs uniformly, without requiring hand-crafted templates or specific HDL constructs.

HARM [17] heavily leverages user-defined hints, which are particularly challenging to define for structural Verilog. The resulting properties are validated against simulation traces. The closest piece of work to ours is ARTmine [23], which focuses on temporal assertion generation by mining patterns. GoldMine was extended by Liu *et al.* [30] to support word-level properties, but still requires pre-defined templates. SMART is the only tool that supports bitvectors, does not require templates, and verifies the generated assertions.

There has also been interest in relying on machine learning models to automatically generate SVAs. Exploiting natural language processing models for automated SVA generation has been widely studied [20], [28], [46], [29], [15], [27], [35], [3]. There has also been interest in using large language models (LLMs) for SVA mining [26], [34], [42], [14]. However, all these approaches require natural language specifications as inputs, while SMART automatically produces SVAs from traces.

Adjacent research in temporal logic mining and reactive synthesis also highlights the contrast with SMART. UN-DINE [12] mines security-critical temporal logic (LTL) properties from RTL designs using typed event templates. While it supports expressive temporal operators, it relies on pre-defined templates and extensive trace preprocessing. SMART avoids templates altogether and synthesizes bit-precise safety assertions for both behavioral and structural Verilog.

Oracle-Guided Inductive Synthesis has been used for de-obfuscating low-level bit-code code [24]. Here, the user must

provide a library of components from which the deobfuscation is constructed, and the low-level code must be small enough that it can be reasoned about by a satisfiability solver. The fact that many of our mutation tests time-out demonstrates that the Verilog files we are able to handle are beyond the scope of these symbolic encoding-driven techniques.

The closest application of oracle-guided inductive synthesis to ours is inductive invariant synthesis [16]. Specification mining is related to but distinct from inductive invariant synthesis, where the user provides a target property, and an algorithm is used to infer an invariant that can be used to construct a proof-by-induction to show the target property holds. Unlike SMART, without a target property, the approaches in the literature for inductive invariant synthesis would not be able to infer an invariant.

A synthesis framework integrating reactive synthesis with SyGuS has been proposed [10], using Temporal Stream Logic modulo theories (TSL-MT) to generate control/data-reactive programs. While both approaches use SyGuS, SMART focuses on mining state-based assertions from execution traces, rather than synthesizing reactive strategies. As such, their synthesis goals and application domains are fundamentally different.

X. CONCLUSIONS

We have presented SMART: a novel approach for hardware specification mining based on oracle-guided synthesis. Our evaluation demonstrates that specifications generated by SMART can differentiate more mutants on real-world hardware verification benchmarks from the literature, using fewer assertions than the state of the art.

REFERENCES

- [1] Ebmc. <https://github.com/diffblue/hw-cbmc/>
- [2] Circuit netlist benchmarks. <https://sportlab.usc.edu/~msabrishami/benchmarks.html> (2025)
- [3] Aditi, F., Hsiao, M.S.: Hybrid rule-based and machine learning system for assertion generation from natural language specifications. In: 2022 IEEE 31st Asian Test Symposium (ATS). pp. 126–131. IEEE (2022)
- [4] Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: POPL. pp. 4–16. ACM (2002)
- [5] Ashenden, P.J.: The designer's guide to VHDL. Morgan kaufmann (2010)
- [6] 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
- [7] Brglez, F., Bryant, D., Kozminski, K.: Combinational profiles of sequential benchmark circuits. In: 1989 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1929–1934. IEEE (1989)
- [8] Chakraborty, R.S., Bhunia, S.: Hardware protection and authentication through netlist level obfuscation. In: 2008 IEEE/ACM International Conference on Computer-Aided Design. pp. 674–677. IEEE (2008)
- [9] Chen, C., Hu, G., Zuo, D., Yu, C., Ma, Y., Zhang, H.: E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. pp. 1–6 (2024)
- [10] Choi, W., Finkbeiner, B., Piskac, R., Santolucito, M.: Can reactive synthesis and syntax-guided synthesis be friends? In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 556–571 (2022)
- [11] Clarke, E.M.: Model checking - my 27-year quest to overcome the state explosion problem. In: LPAR. Lecture Notes in Computer Science, vol. 5330, p. 182. Springer (2008)
- [12] Deutschbein, C., Sturton, C.: Mining security critical linear temporal logic specifications for processors. In: 2018 19th International Workshop on Microprocessor and SOC Test, Security and Verification (MTV). pp. 13–18. IEEE (2018)
- [13] Eisner, C., Fisman, D.: A practical introduction to PSL. Springer Science & Business Media (2007)
- [14] Fang, W., Li, M., Li, M., Yan, Z., Liu, S., Xie, Z., Zhang, H.: Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms (2024), <https://arxiv.org/abs/2402.00386>
- [15] Frederiksen, S.J., Aromando, J., Hsiao, M.S.: Automated assertion generation from natural language specifications. In: 2020 IEEE International Test Conference (ITC). pp. 1–5. IEEE (2020)
- [16] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 69–87. Springer (2014)
- [17] Germiniani, S., Pravadelli, G.: Harm: a hint-based assertion miner. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 4277–4288 (2022)
- [18] Godbole, A., Ye, L., Manerkar, Y.A., Seshia, S.A.: Modelling and verification of security-oriented resource partitioning schemes. In: FMCAD. pp. 268–273 (2023)
- [19] Hansen, M.C., Yalcin, H., Hayes, J.P.: Unveiling the iscas-85 benchmarks: A case study in reverse engineering. IEEE Design & Test of Computers **16**(3), 72–80 (1999)
- [20] Harris, C.B., Harris, I.G.: Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 966–971. IEEE (2016)
- [21] Hertz, S., Sheridan, D., Vasudevan, S.: Mining hardware assertions with guidance from static analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32**(6), 952–965 (2013)
- [22] IEEE: Combining dynamic slicing and mutation operators for ESL correction (2012)
- [23] Iman, M.R.H., Jervan, G., Ghasempouri, T.: Artmime: Automatic association rule mining with temporal behavior for hardware verification. In: 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2024)
- [24] Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (1). pp. 215–224. ACM (2010)
- [25] Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017)
- [26] Kande, R., Pearce, H., Tan, B., Dolan-Gavitt, B., Thakur, S., Karri, R., Rajendran, J.: Llm-assisted generation of hardware assertions. arXiv preprint arXiv:2306.14027 (2023)
- [27] Keszocze, O., Harris, I.G.: Chatbot-based assertion generation from natural language specifications. In: 2019 Forum for Specification and Design Languages (FDL). pp. 1–6. IEEE (2019)
- [28] Krishnamurthy, R., Hsiao, M.S.: Controlled natural language framework for generating assertions from hardware specifications. In: 2019 IEEE 13th International Conference on Semantic Computing (ICSC). pp. 367–370. IEEE (2019)
- [29] Krishnamurthy, R., Hsiao, M.S.: Ease: Enabling hardware assertion synthesis from english. In: Rules and Reasoning: Third International Joint Conference, RuleML+ RR 2019, Bolzano, Italy, September 16–19, 2019, Proceedings 3. pp. 82–96. Springer (2019)
- [30] Liu, L., Lin, C.H., Vasudevan, S.: Word level feature discovery to enhance quality of assertion mining. In: Proceedings of the International Conference on Computer-Aided Design. pp. 210–217 (2012)
- [31] Lo, D., Khoo, S.: Mining patterns and rules for software specification discovery. Proc. VLDB Endow. **1**(2), 1609–1616 (2008)
- [32] Neider, D., Roy, R.: What is formal verification without specifications? A survey on mining LTL specifications. In: Principles of Verification (3). Lecture Notes in Computer Science, vol. 15262, pp. 109–125. Springer (2024)
- [33] Niemetz, A., Preiner, M.: Bitwuzla. In: International Conference on Computer Aided Verification. pp. 3–17. Springer (2023)
- [34] Orenes-Vera, M., Martonosi, M., Wentzlaff, D.: Using llms to facilitate formal verification of rtl. arXiv e-prints pp. arXiv-2309 (2023)

- [35] Parthasarathy, G., Nanda, S., Choudhary, P., Patil, P.: Spectosva: Circuit specification document to systemverilog assertion translation. In: 2021 Second Document Intelligence Workshop at KDD (2021)
- [36] Raveendran, R., Bhuinya, S.: Customization of ibex risc-v processor core. Customization of Ibex RISC-V Processor Core (2021)
- [37] Rosser, B.J.: Cocotb: a python-based digital logic verification framework. In: Micro-electronics Section seminar. CERN, Geneva, Switzerland (2018)
- [38] Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a sat-solver. In: International conference on formal methods in computer-aided design. pp. 127–144. Springer (2000)
- [39] Shin, H.: Data-centric machine learning pipeline for hardware verification. In: 2022 IEEE 35th International System-on-Chip Conference (SOCC). pp. 1–2. IEEE (2022)
- [40] Shin, H.: Efficient bug discovery with machine learning for hardware verification. <https://community.arm.com/arm-research/b/articles/posts/efficient-bug-discovery-with-machine-learning-for-hardware-verification> (2025)
- [41] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415. ACM (2006)
- [42] Sun, C., Hahn, C., Trippel, C.: Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions. In: First International Workshop on Deep Learning-aided Verification (2023)
- [43] Thomas, D., Moorby, P.: The Verilog® hardware description language. Springer Science & Business Media (2008)
- [44] Vijayaraghavan, S., Ramanathan, M.: A practical guide for SystemVerilog assertions. Springer Science & Business Media (2005)
- [45] Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* **5**(POPL), 1–29 (2021)
- [46] Zhao, J., Harris, I.G.: Automatic assertion generation from natural language specifications using subtree analysis. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 598–601. IEEE (2019)

Automated Formal Verification of a Software Fault Isolation System

Matthew Sotoudeh* , Zachary Yedidia* 

*Stanford University, Stanford, USA
 {sotoudeh, zyedidia}@stanford.edu

Abstract—Software fault isolation (SFI) is a popular way to sandbox untrusted software. A key component of SFI is the *verifier* that checks the untrusted code is written in a subset of the machine language that guarantees it never reads or writes outside of a region of memory dedicated to the sandbox. Soundness bugs in the SFI verifier would break the SFI security model and allow the supposedly sandboxed code to read protected memory. In this paper, we address the concern of SFI verifier bugs by performing an automated formal verification of a recent SFI system called Lightweight Fault Isolation (LFI). In particular, we formally verify that programs accepted by the LFI verifier never read or write to memory outside of a designated sandbox region.

Index Terms—Verification, software fault isolation

I. INTRODUCTION

Computer systems frequently need to run untrusted code, such as web browsers running applets and scripts, software linking against third-party libraries, or operating systems accepting user code into the kernel for packet filtering. In all of these cases, security concerns demand that the host application only run the untrusted code in ways that guarantee the untrusted code will not read from or write to the host application’s memory. In theory, the untrusted code could be run in an entirely separate address space (process) from the host code. In practice, processes introduce far too much context switching overhead to be reasonable for many applications. Requiring the untrusted code to be written in a memory-safe language like Java, JavaScript, EBPF, or WebAssembly that can be executed in the same address space improves startup time but is significantly slower than native machine code. Furthermore, it makes the entire language interpreter or compiler a security-critical part of the trusted codebase.

This paper focuses on the *software fault isolation* (SFI) mechanism of software sandboxing. In SFI, the untrusted code is compiled to a memory-safe subset of the machine code (ARM64). That binary is only executed if it passes a *SFI verifier*, which checks that all memory access instructions in the binary target memory within the sandbox. Of course, verifying arbitrary machine code is hard; in practice, SFI techniques require that the guest code be compiled to use a restricted subset of memory operations that are easy to verify (a naïve SFI scheme, for example, might require every memory read or write be immediately preceded by a bounds check).

While SFI is a popular technique with many different instantiations [1]–[3] we focus on the recent Lightweight Fault Isolation (LFI) system [3]. LFI is a very low-overhead sandboxing system specialized for ARM64. There is ongoing work to bring LFI to production use-cases, and large-scale

deployment motivates extra scrutiny on the LFI verifier, which is tasked with the security-critical job of verifying that a given binary cannot access memory outside of its dedicated sandbox.

We discuss how the LFI verifier accomplishes this in Section II, but briefly, it involves a whitelisted subset of ARM64 instructions that are chosen to maintain certain memory isolation guarantees. However, the ARM64 includes a very large number of instructions, and it is conceivable that those in the whitelist interact in nontrivial ways to break out of the sandbox. Indeed, earlier versions of LFI have had soundness bugs because we did not realize that certain whitelisted instructions could modify registers in unexpected ways.

This paper describes our work on **automatic formal verification of the lightweight fault isolation SFI system** for the ARM64 architecture; the LFI verifier is now equipped with a formal proof that every program it accepts cannot read outside of its dedicated sandbox memory region.

Our heavy automation is unique among existing formally verified SFI systems [2]. The only manual intervention needed is to write down an *SFI invariant*, which requires only about 20 lines of SMT-LIB2 code. To enable this, we codesigned the SFI system and verification pipeline. Traditional SFI systems frequently check for the presence or absence of *sequences* of instructions and sometimes do complicated dataflow analyses to ensure memory isolation. But these nontrivial analyses are very difficult to formally prove precise facts about. Instead, we realized that the LFI scheme is essentially a *stateless* SFI system, i.e., its verifier is nothing more than an instruction whitelist. The guest program may contain any instruction in the whitelist, in any order, and no other instructions. This means we need merely check that each whitelisted instruction, on its own, maintains the expected SFI invariants.

The rest of this paper summarizes the LFI system (Section II), describes our verification pipeline (Section III), and summarizes future and related work (Sections IV and V). An archival version of our source code and tools is available in [4].

II. BACKGROUND ON LIGHTWEIGHT FAULT ISOLATION

This section briefly describes the LFI system for software fault isolation. Consider a setting where a web browser downloads and wants to execute untrusted machine code from the Internet (similar to the NaCl model used by Chromium). In this setting, the website operator providing the machine code will compile it using an untrusted, but LFI-aware compiler into ARM64 binary code. This binary code is then sent to the browser, who wishes to run it while having some guarantee

that it will not perform unauthorized memory operations. To do so, the browser provides the ARM64 binary to the *LFI Verifier*, which is essentially a sound program analyzer for memory isolation. If the verifier rejects the program, it might be unsafe and so the browser refuses to run it. On the other hand, if the verifier accepts the program, it is guaranteed to be memory isolated and so the browser can confidently run it. The program might make calls into the LFI runtime to request actions like drawing to the screen or sending network packets.

Safety relies on two crucial aspects: the LFI verifier and the LFI runtime. A buggy verifier might accept code that reads from the host computer’s memory in insecure ways, while a buggy runtime might allow the binary to manipulate it into performing unpermitted operations.

This paper considers only the LFI verifier, and so we assume the LFI runtime is implemented correctly. The notion of memory isolation here is more coarse grained than memory safety in traditional programming languages: the untrusted program may access any part of a 4GiB ‘sandbox’ devoted to it. It can do anything in that sandbox; it might interpret portions of it as arrays and then index those application-interpreted arrays out-of-bounds, as long as it never successfully¹ reads or writes outside of its 4GiB sandbox.

The rest of this section briefly describes how the LFI verifier checks memory dereferences in its simpler sparse mode; complete details on the LFI system are available in [3].

A. Reserved Registers for Memory Isolation

The job of the LFI verifier (checking the given code is memory isolated) is, in full generality, undecidable. Hence, it must check instead for a subset of known-to-be-memory-safe constructs (sound but incomplete). The set of constructs it checks for must allow efficient compiled code.

The primary trick is *reserved registers*. There is a reserved *sandbox base register* (x21 in LFI) pointing to the first byte of the program’s 4GiB sandbox region. This region is assumed to be 4GiB aligned, so the bottom 32 bits of the x21 register are zero. There is also a reserved *addressing register* (x18 in LFI) which the verifier ensures is the only register (with some exceptions) used for the ARM64 load and store instructions. To a first approximation, the LFI verifier iterates over the binary, disassembles each instruction, and checks:

- No instruction ever writes to x21, so x21 will always contain the sandbox base address.
- The only instructions that can write to x18 are those of the form `add x18, x21, wN, uxtw`, which results in x18 containing the high 32 bits of register x21 concatenated to the low 32 bits of register xN.
- The only instructions that can load from or store to memory use x18 as the memory address.

¹Notably our verification only shows that the program never *successfully* reads or writes outside of the sandbox; valid LFI programs can sometimes make *attempted* reads or writes as long as those attempts are guaranteed to terminate execution, e.g., they access a page that the runtime guarantees is unmapped. This is discussed further in Section II-B.

A program that passes this check is thus unable to explicitly load memory outside of the 4GiB sandbox region, because it can only load from x18 and x18 can only be set by an operation that ensures its final value is within the sandbox region (i.e., has the same upper 32 bits as x21).

Note that, beyond this simple approximation, there are many corner cases that need to be handled [3]. For example, x18 might be set to the very last byte in the 4GiB sandbox region, but then doing an 8 byte load from that address will actually read outside of the 4GiB sandbox region. To resolve this, the LFI specification [5] requires that the sandbox be surrounded by unmapped ‘redzone regions’ so that such corner case reads cause a trap rather than reading host memory. Use of the stack pointer register presents other complications, described below.

B. Handling the Stack

In addition to direct memory loads via the x18 register, LFI also allows loads relative to the stack pointer (sp). This introduces extra complications as *writeback loads* are used to simulate a stack push. For example, the instruction

```
// sp = sandbox_base
ldr x0, [sp], #-8
// sp = sandbox_base - 8
```

dereferences the stack pointer and then decrements it by 8 bytes. But if the stack pointer is initially pointing at the base of the sandbox region, after executing this instruction it will now be pointing outside of the sandbox region. To resolve this, the LFI runtime guarantees that 4GiB of address space preceding and following the sandbox region is unmapped, so attempting to actually access this region would cause a trap and hence safe exit of the guest program (the largest offset accepted by the writeback load operation is smaller than 4GiB).

The reader might be concerned that a chain of such instructions might leave sp past even this 4GiB buffer region and hence possibly pointing at host memory:

```
// sp = sandbox_base
ldr x0, [sp], #-8
// sp = sandbox_base - 8
ldr x0, [sp], #-8
// sp = sandbox_base - 16
ldr x0, [sp], #-8
...
// sp = sandbox_base - 4GiB - 8?
ldr x0, [sp], #-8 // load bad data?
```

But indeed this is not possible because the second operation in this sequence would trap (accessing unmapped memory), transfer control back to the LFI runtime, and thus terminate the execution of the sandboxed code.

C. Compiler Instrumentation to Generate LFI-Passing Code

Compilers rarely produce code passing the LFI verifier, e.g., compiled code often loads from a register other than x18 or writes to x18 using a nonwhitelisted instruction. Hence, the LFI project includes a modified compiler toolchain that

produces LFI-passing binary. To do so, it tells the compiler to reserve registers $x18$ and $x21$. Then, after the compiler produces an assembly file, we identify instructions that would fail LFI verification and rewrite them into sequences of instructions that pass LFI verification. For example, loading from $x5$ may not be safe because the LFI verifier has no way of knowing that it points into the sandbox region:

```
ldur x2, [x5]
```

Instead, the rewriter might transform this into the two-operation sequence:

```
add x18, x21, w5, uxtw
ldur x2, [x18]
```

Which replaces the upper 32 bits of $x5$ with those of $x21$ before the load. If $x5$ was in-bounds at this point in the program execution, then this transformation has no effect. But if $x5$ was out-of-bounds, this transformation causes it to load an in-bounds address instead. This may lead to an application-level bug in the untrusted code, but effectively prevents it from accessing the host memory.

III. VERIFICATION OF LFI

This section describes how we verify correctness of the LFI system. Recall that the LFI system is a whitelist of allowed instructions and a set of memory mappings guaranteed by the runtime (e.g., that the page following the sandbox region is trap-on-execute). There are actually two variants of LFI that we verify: sparse LFI (where every sandbox is surrounded by 4GiB redzone regions of unmapped memory) and dense LFI (where sandboxes are packed more tightly, with internal unmapped regions). In this section we focus on verification of sparse LFI; dense LFI is verified similarly. In general, we verify the following:

If the registers satisfy a certain *SFI invariant* I , and the next instruction to be executed is one of the whitelisted instructions, then when executing the instruction both (1) no memory operations (reads, writes, executes) outside of the 4GiB sandbox succeed; and (2) either the execution terminates or the SFI invariant I holds on the successor state as well.

This is verified per instruction: so as long as the guest code starts in a state satisfying the SFI invariant I and all of the instructions in the sandbox region satisfy that property, it will continually satisfy that invariant during the execution of the guest program and not make any unsafe memory operations until it terminates. Execution can terminate by executing an instruction that causes a hardware trap (like reading from memory that is unmapped or executing an undefined instruction) or by calling in to the LFI runtime.

A. Register Invariants

In the sparse LFI variant, we verify the following invariants:

- $x21$ points to the special 4GiB-aligned sandbox base address, which does not change during execution.

- $x18$ and the stack pointer are both in the range $[x21 - 128\text{MiB}, x21 + 4\text{GiB} + 128\text{MiB})$.²
- The program counter (PC) is in the range $[x21, x21 + 4\text{GiB})$.
- R_{30} is either in the range $[x21, x21 + 4\text{GiB})$, or contains one of three special LFI runtime call addresses.

Note the second of these is to handle the case of writeback load operations as described in Section II-B.

B. Detecting Bad Side Effects

The main goal of our verification is to ensure that the guest never breaks out of its sandbox. We record any (symbolic) memory reads or writes performed during the (symbolic) execution of the instruction. We say a bad side effect occurs if any of them successfully accesses an address outside of the 4GiB sandbox region (note that accesses guaranteed to fail are not successful; e.g., reading from a page that the runtime guarantees is unmapped terminates execution of the sandboxed program). An instruction fails verification if it can ever perform a bad side effect when starting in a CPU state that satisfies the invariants and expected memory mappings.

C. Memory Mapping Assumptions and Detecting Faults

Per the LFI specification, we assume that the LFI runtime has set up the following memory mappings:

- The 4GiB regions before and after the sandbox 4GiB region are both unmapped, so accesses to them trap.
- The first 4KiB of the sandbox region is marked read-only, and the first three words at that location contain the addresses of special LFI runtime calls (analogous to system calls).

D. Handling Faults

We need to detect when the program execution terminates. We call this a ‘fault,’ and it can occur in one of a few ways:

- The PC (before or after the instruction) is not 4 byte aligned or is in a region of memory that we know is mapped no-execute.
- A memory read or write occurs to an address that we know is unmapped.
- The PC becomes one of the three special trampoline addresses.
- An explicit trap occurs, e.g., an undefined instruction.

Some nuance arises here because, to the best of our knowledge, ARM does not always make strict guarantees about the order of execution of traps. Consider a write to a memory range crossing two bytes, b and $b+1$. If b is unmapped memory but $b+1$ is host memory outside of the sandbox region, the operation is safe only if the trap is guaranteed to occur *before* the $b+1$ write happens, i.e., if the hardware bounds checks ‘eagerly.’ Rather than assume that the ordering of these checks is guaranteed or deterministic across microarchitectures, we check a much stronger property: bad side effects do not occur *even if the traps always happen last*.

²In dense mode, this is $\pm 8\text{KiB}$.

E. Implementation and Formal Model of ARM64

Critical to our approach is the ability to, given an instruction encoding and a desired logical property about the CPU state before and after execution of the instruction, check whether that property holds. This requires a formal model of the CPU semantics. For this, we relied on the ASL formal specification of ARM64 [6]. In particular, we used the ASLP interpreter [7] which can partially evaluate the ASL specification and returns a symbolic intermediate representation that summarizes the effects of the instruction on a symbolic model of the CPU.

We wrote a Python library that extracts this symbolic intermediate representation from the ASLP interpreter into a usable Python form. We then wrote a symbolic execution engine for this intermediate representation. The output of the symbolic execution is a series of SMT-LIB2 assertions encoding the execution of the instruction. After adding assertions checking the SFI invariant, we use the `Yices2` SMT solver to determine satisfiability [8].

Writing the symbolic execution engine was simpler than it sounds, because we only need it to be sound, not complete. In particular, because so few of the operations actually affect the reserved registers, we can leave many functions uninterpreted.

We ran into two issues with the ASL specification and ASLP partial evaluator that required extra work. First, the ASL spec and ASLP partial evaluator lacked proper support for expressing atomic instructions, which led to internal ASLP exceptions when partially evaluating those instructions. To solve this, we commented out the atomics-related ASL code that was causing the issues, as our verification is not interested in race conditions. Second, the ASL spec’s handling of system access traps returns a special record type which also causes internal ASLP errors. We resolved this by replacing the system access trap generation functions with an assignment to a special `SYMEX_DID_TRAP` variable that indicates to our symbolic execution engine directly when a trap occurs.

F. Parallel Verification and Results

Our verification technique involves iterating through all whitelisted ARM64 instructions. To do so, we first iterate through all of the ARM64 instruction set (2^{32} total instructions) and check with the stateless LFI verifier for which ones were accepted. We evenly divide those accepted instructions across all of the verifier worker threads.

We ran verification in parallel on a 160-core Ampere Altra ARM64 machine. Verification for each of the sparse and dense LFI systems required approximately 20 hours each. Throughout the verification process we identified some inaccuracies and missing details in the LFI specification [5], including how LFI runtime calls are handled and memory mapping guarantees for the first and last sandboxes. We are in the process of correcting these oversights in a future version of the specification, but these are only missing details in the written specification. After clarifying these specification issues, the verification process completed and found no bugs.

IV. LIMITATIONS AND FUTURE WORK

This section describes possible future work. First, we could use our verification pipeline to automatically synthesize SFI systems from their invariants by iterating over all 2^{32} ARM64 instructions and including only those that maintain the invariants. This could enable rapid experimentation with SFI systems, especially to different domains like determinism.

Second, we could try to more fully automate our verification pipeline by using invariant inference techniques to automatically derive the invariant for a given SFI system.

To speed up our verification pipeline we could also check groups of instructions at once, e.g., all add instructions targeting a nonreserved register. This would require the ability to extract shared semantics for groups of instructions or instruction encodings with symbolic bits. In theory the ASLP interpreter supports this [7] but we decided to use the simpler strategy for this paper and found performance was reasonable.

This work did not consider safety of the LFI runtime, which remains an unverified but trusted component of the system.

In this work we focused on verifying an SFI system for the ARM64 architecture. But LFI also supports the x86-64 architecture, and there has been recent work attempting to extend it to RISC-V. It would be interesting to produce a fully verified SFI system for x86-64 or RISC-V. We believe that the verification strategy in this paper would work well for verifying the RISC-V version of LFI, which involves a similarly sized set of whitelisted instructions and is also effectively stateless. However, making the x86-64 LFI system stateless requires recognizing a number of ‘macro-operations’ that are actually sequences of other instructions (similar to `nacljump` in the original NaCL paper [1]). This leads to the set of whitelisted (macro)instructions for x86-64 LFI being much larger than the set for ARM64 and RISC-V, making it infeasible to enumerate through all of them. In theory, as described earlier, a verification pipeline that lets us leave some bits (e.g., immediate values) symbolic would make verifying x86-64 LFI feasible as well.

V. RELATED WORK

a) Software Fault Isolation: Pittsfield was an early SFI system [9]. The Native Client (NaCl) SFI system was shipped with Google Chrome and used for website-loaded add-ons such as required (at the time) by Netflix [1]. This shows the security critical position that SFI systems play, motivating the use of formal verification to ensure the absence of bugs. More recently, the LFI system has been proposed as an optimized SFI system for ARM64.

To our knowledge, ours is the first stateless SFI system. Native Client, for example, required the use of a special two-instruction ‘`nacljump`’ sequence of instructions to perform indirect jumps while maintaining control flow integrity. Automated verification of such a stateful SFI verifier would be more complicated, because we can no longer look at each instruction on its own, leading to an impractically large encoding space of instructions to automatically iterate over. Similarly, LFI as proposed in [3] allows some within-basic-block optimizations,

particularly removing guards to stack pointer modification when a stack access is performed later in the basic block. Since then, we have found those optimizations can be removed with minimal performance impact, making it fully stateless.

b) Formal Models of ISAs: This work used the ARM64 formal specification extracted by the ASLP [7] partial evaluator. [10] and [11] are other formal specifications of hardware ISAs. At the beginning of this project we tried to use the SAIL [11] architecture semantics. In theory, using SAIL would have made it easier to extend our technique to architectures beyond ARM64, since SAIL contains models of multiple different architectures. However, we found that the SAIL models simulate the architecture at a lower level than we needed. Furthermore, because the SAIL models were so fine-grained, they required significantly more investment for us to understand the model and even significantly more time to just run the compilers necessary to work with the model. Because our verification technique involves running a symbolic execution of the architecture semantics for every one of the LFI-whitelisted instructions, we decided to instead go for the ASLP semantics which were both simpler for us to work with and had faster tooling.

c) Verified Sandboxing: While we believe ours is the first *automatically* verified SFI system, there have been formally verified sandboxing systems before. VeriWasm [2] has an interactively verified SFI system used as a WebAssembly target. vWasm [12] is a verified compiler from WebAssembly to x86-64 machine code written in F*. WAVE [13] verifies a WASM runtime. RockSalt [14] and ARMor [15] use interactive theorem provers to verify classic SFI verifiers.

d) Verified Program Analyses: The LFI verifier is itself essentially a program analyzer. The problem of verifying program analyzers has been explored in prior work, such as ALIVE [16] that verifies program analyzers and transformations in a domain-specific language, and ALIVE2 [17] that does bounded translation validation for LLVM optimizations.

ACKNOWLEDGEMENTS

We thank the National Science Foundation (DGE-1656518, DGE-2146755), Stanford IOG Research Hub, and Stanford Future of Digital Currency Initiative for supporting grants.

REFERENCES

- [1] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *30th IEEE Symposium on Security and Privacy (SP 2009)*, 17-20 May 2009, Oakland, California, USA. IEEE Computer Society, 2009, pp. 79–93. [Online]. Available: <https://doi.org/10.1109/SP.2009.25>
- [2] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, “SFI safety for native-compiled wasm,” in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.
- [3] Z. Yedidia, “Lightweight fault isolation: Practical, efficient, and secure software sandboxing,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafir, Eds. ACM, 2024, pp. 649–665. [Online]. Available: <https://doi.org/10.1145/3620665.3640408>
- [4] M. Sotoudeh and Z. Yedidia, “(artifact) automated formal verification of a software fault isolation system,” 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.16883694>
- [5] “LFI specification (draft).” [Online]. Available: <https://www.scs.stanford.edu/~zyedidia/docs/lfi/lfi-spec.pdf>
- [6] A. Reid, “Trustworthy specifications of arm® v8-a and v8-m system level architecture,” in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 161–168. [Online]. Available: <https://doi.org/10.1109/FMCAD.2016.7886675>
- [7] K. Lam and N. Coughlin, “Lift-off: Trustworthy armv8 semantics from formal specifications,” 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. 274–283. [Online]. Available: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_36
- [8] B. Dutertre, “Yices 2.2,” in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49
- [9] S. McCamant and G. Morrisett, “Evaluating SFI for a CISC architecture,” in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, A. D. Keromytis, Ed. USENIX Association, 2006. [Online]. Available: <https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture>
- [10] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, “A complete formal semantics of x86-64 user-level instruction set architecture,” 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. 1133–1148. [Online]. Available: <https://doi.org/10.1145/3314221.3314601>
- [11] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, “ISA semantics for armv8-a, risc-v, and CHERI-MIPS,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, 2019. [Online]. Available: <https://doi.org/10.1145/3290384>
- [12] J. Bosamiya, W. S. Lim, and B. Parno, “Provably-safe multilingual software sandboxing using WebAssembly,” in *Proceedings of the USENIX Security Symposium*, August 2022.
- [13] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, “Wave: a verifiably secure webassembly sandboxing runtime,” in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2940–2955. [Online]. Available: <https://doi.org/10.1109/SP46215.2023.10179357>
- [14] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, “Rocksalt: better, faster, stronger SFI for the x86,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 395–404. [Online]. Available: <https://doi.org/10.1145/2254064.2254111>
- [15] L. Zhao, G. Li, B. De Sutter, and J. Regehr, “Armor: fully verified software fault isolation,” in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 289–298. [Online]. Available: <https://doi.org/10.1145/2038642.2038687>
- [16] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, “Provably correct peephole optimizations with alive,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 22–32. [Online]. Available: <https://doi.org/10.1145/2737924.2737965>
- [17] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr, “Alive2: bounded translation validation for LLVM,” in *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 65–79. [Online]. Available: <https://doi.org/10.1145/3453483.3454030>

Static Coverage in Deductive Software Verification

Aaron Tomb



Amazon, Portland, OR, USA
aarotomb@amazon.com

Anjali Joshi

Amazon, Boston, MA, USA
anjalijs@amazon.com

Abstract—The results of software verification are only as trustworthy as the provided specification. Errors or incompleteness in the specification can result in unwarranted confidence in the implementation. Previous work, particularly in the realm of model checking, has investigated a notion of *coverage* in verification. Portions of the system that could be replaced arbitrarily without causing verification failure are considered uncovered. We show that the same notion of coverage used in model checking can be applied to deductive software verification, with a reasonable performance penalty when using an implementation based on unsatisfiable cores. This approach provides opportunities for identifying specification gaps by detecting vacuous proofs, unconstrained code, and unnecessary or redundant specifications. We describe an implementation of this approach for the Boogie intermediate verification language, the use of this implementation from the verification-aware programming language Dafny, and experimental results on a large corpus of industry-scale Dafny code.

I. INTRODUCTION

Verification provides a high degree of assurance that software behaves as *specified*. Assurance that the software behaves as *intended* depends on the correctness and completeness of the specification — how well it agrees with the designer’s goals. Although techniques to detect specification problems do exist in software verification, most focus entirely on detecting vacuous proofs, and most adopt a somewhat ad-hoc technique (most commonly, attempting to prove a contradiction at various points throughout the program). Prior work in the model checking community formalized a notion of the coverage of a verified program [22], saying that a transition system element is covered if modifying it could possibly cause verification to fail. Further work showed that coverage is dual to vacuity [21], and therefore that coverage analysis can be used to detect vacuity.

In this paper, we extend this prior work and apply it in the context of deductive software verification that supports separate specification and implementation constructs. We describe a technique for static coverage that in addition to detecting vacuous proofs, also identifies redundant specifications and unconstrained portions of the implementation. Each program construct that is not covered is relevant for identifying a specific type of specification gap:

- uncovered goals are proved vacuously,
- uncovered assumptions are unnecessary, and
- uncovered implementation statements are either unconstrained by the specification or unreachable.

We describe the technique for computing static coverage in deductive verification and an implementation of this technique

within the Dafny verification-aware programming language [25] via the Boogie intermediate verification language [27]. Our experimental results evaluating this implementation demonstrate modest computational overhead relative to verification, making it practical to employ as integral part of verification. We also show how this technique can be used to generate actionable warnings without false positives.

A. Examples

Before exploring the technique in detail, we first consider concrete examples of these specification problems, such as the Dafny program in Fig. 1.

```

1  function {:axiom}
2  Find(xs:seq<int>, x:int,
3      beg:int, end:int): (i:int)
4      requires 0 <= beg < |xs| &&
5              0 <= end < |xs|
6      ensures beg < i < end
7      ensures xs[i] == x
8
9  method CallFind() {
10     var xs := [42, 43];
11     var i := Find(xs, 47, 0, 1);
12     assert xs[i] == 45;
13 }
```

Fig. 1. Dafny code with a subtly contradictory specification. The postcondition on line 6 becomes unsatisfiable when instantiated with the concrete arguments passed in on line 11.

In this code, the `Find` function is specified but not implemented, as may be the case partway through the process of implementing a verified program, and the `{:axiom}` annotation emphasizes that its contract is assumed to hold. This function is intended to search for a given element, `x`, between indices `beg` and `end` of the sequence `xs`. For this to work, it requires both indices to be within range, and ensures that the returned index is between the provided indices. This seems reasonable at first, but there is a mistake in the postcondition. If `end` is `beg + 1`, no return value `i` is possible. In the `CallFind` method, the passed indices are adjacent, so the postcondition of `Find` is false. Because of this, Dafny can prove the clearly false assertion at the end of the method. However, by default, Dafny does nothing to indicate the fact that this proof succeeds due to an inconsistency. As we will

show, computing coverage will allow detection of the fact that this assertion was proved vacuously, with no additional effort from the programmer.

Similarly, consider the Dafny program in Fig. 2 that has a redundant specification. One could imagine believing (incorrectly, but perhaps reasonably) that the index bound safety of the standard implementation of binary search requires the array to be sorted, and therefore providing the precondition shown. Dafny would accept this without complaint and continue to prove that the implementation does not use out-of-bounds array indices (an implicit specification that Dafny attempts to prove of every program). Dafny would not, by default, do anything to indicate that this precondition is unnecessary. Computing coverage will allow detection of this redundancy.

```

1  method BinSearch(a:seq<int>, k:int)
2      returns (r:int)
3      requires forall i,j ::
4          0 <= i < j < |a| ==>
5          a[i] <= a[j]
6  {
7      var lo, hi := 0, |a|;
8      while lo < hi
9          invariant 0 <= lo <= hi <= |a|
10     {
11         var mid := (lo + hi) / 2;
12         if k < a[mid] {
13             hi := mid;
14         } else if a[mid] < k {
15             lo := mid + 1;
16         } else {
17             return mid;
18         }
19     }
20     return -1;
21 }
```

Fig. 2. Dafny code with an unnecessary precondition (lines 3, 4, 5) and code that is not constrained by the specification (lines 17, 20).

Finally, consider the specification of the same binary search implementation. Some of the code is necessary to ensure array bounds safety, such as the calculation of `mid`. However, other portions of the code, such as the `return` statements, could be changed arbitrarily without affecting the results of verification.

Computing coverage allows more useful verifier feedback for each of these examples than previously available, and more granular information than available from previous verification coverage analysis. This approach is not specific to Dafny, and could be adopted by any verifier based on weakest preconditions.

B. Overview of Paper

Our contributions beyond prior work are as follows:

- We formalize how coverage translates to deductive verification, specifically the verification condition (VC) generation process, using unsatisfiable cores.
- We define the type of specification gap implied by each type of uncovered element. Our interpretation allows automatic detection of:
 - contradictory assumptions or branches, leading to vacuous proofs;
 - redundant specifications that can be removed; and
 - unconstrained implementation code that could be specified or removed.
- We provide experimental results showing reasonable computational overhead and useful warnings on a large corpus of real-world Dafny code.

We first review prior work in more detail. Then we define more precisely what coverage means in the abstract, and how it applies to compositional verification of programs with contracts (as opposed to monolithic transition systems), before detailing how to calculate coverage. Finally, we describe how we implemented the approach in Dafny and Boogie, and provide measurements of running the analysis on several Dafny code bases, including performance, warning count, and warning classification.

II. PRIOR WORK

The term *coverage* has been used since at least the 1960s [10], usually to describe the set (or percentage) of syntactic program elements executed by a test or set of tests. (Although some cases, such as path coverage and MCDC extend beyond program elements into more semantic notions.) We describe a notion of coverage intended to be compatible with that used in testing, but that captures the elements that took place in a proof.

Le, *et al.* [23], investigated the idea of combining coverage from proofs and tests. Their definition of coverage is based on the degree to which mutations to the program can cause proof or test failure. Like our work, they consider coverage in terms of the syntactic structure of the program. They focus on how to combine proof and test coverage, rather than on how to calculate either, though they imply the use of mutation testing.

In the verification community, significant research has gone into the problem of determining how well a specification describes an implementation, the most relevant being connected to one of (a) deductive software verification using SMT or first-order logic, (b) model checking with respect to temporal logic specifications, or (c) analysis of high-level modeling languages.

In the realm of deductive software verification, one common approach to identifying either specification problems or problems in unspecified programs is to do some form of reachability analysis to identify infeasible portions of code or specification in the context of any available assumptions [11], [19], [20], [29].

One variation of this approach, sometimes referred to as “smoke testing”, is available in several verification tools, including Boogie [5], [24], [27], Why3 [4], [31], and Frama-C [2], [13]. This approach involves adding `assert false`

statements in relevant locations. Any that are provable indicate either an inconsistent specification or dead code.

These smoke testing features address part of the problem we aim to solve, but somewhat less efficiently, and do not attempt to identify how much of a program is covered by the given specification, whether any parts of the specification are redundant, or any other issues with specification quality.

The model checking community has extensively explored the problem through the lens of coverage, concluding that coverage and vacuity are dual notions [21]. According to this duality, the question is to determine, given a system that successfully verifies, what parts can change while still allowing verification to succeed. If a portion of the implementation can change arbitrarily without causing verification to fail, then either that portion of the implementation is not covered by the specification or some portion of the specification is vacuous, allowing any implementation. If part of the specification can change without causing verification to fail, then that part of the specification is unnecessary for proving the goal. Generally, such portions of the specification can be removed.

The original work describing this duality assumed the use of mutation testing as a mechanism for evaluating coverage or vacuity [22]. By changing a program element and attempting to re-prove the program, it's possible to determine whether that element is necessary for the proof.

As a more efficient alternative, one can do a proof that the implementation satisfies the specification and identify the facts used to complete the proof, which can be done in several ways. When using an SMT solver, the clauses that appear in an unsatisfiable core provide an overapproximation of one possible set of facts needed to complete a proof. The VagTree tool uses this approach to do two types of vacuity detection during bounded model checking, and extracts even more precise information by examining the resolution proof required to show that the core is unsatisfiable [28]. The related VagUoT tool extracts similar information from BDD-based proofs [17]. Later work described the generalization of unsatisfiable cores to unbounded model checking, as inductive validity cores [15], and described how to use these to measure the portion of an implementation covered by a specification [16].

The work on bounded and unbounded model checking was based on transition systems rather than programming languages, however. Unlike deductive software verification, these proofs typically aren't "modular" in the sense that each completely describes the correctness of an element of code. We describe how to take a similar approach in the context of deductive verification of software, along with the opportunities that arise in the new context.

Working with declarative models rather than a combination of a specification and an implementation, Torlak *et al.*, describe an approach for calculating coverage of Alloy specifications using unsatisfiable cores [30], with the goal of helping detect mistakes.

Finally, Beckert, *et al.* [3] addressed the problem of analyzing incomplete proofs to determine how much of the program they had successfully verified. This work calculated coverage

as a subset of the state space of a program rather than a subset of its structural elements. It diverges from our work in two dimensions: first by considering failed proofs and second by considering the semantic state space rather than syntactic elements.

III. VERIFICATION COVERAGE

A. Definition

We define *verification coverage* as a property of programs that meet their specifications as shown by a verifier. As in previous model checking work, we define the covered elements of the program (and its specification) to be those that, if changed, *could* cause verification to fail. Not all changes would necessarily lead to failure, but for each covered element there is at least one change that could lead to the program no longer verifying. Conversely, uncovered elements are those that could be changed arbitrarily (within the bounds of a well-formed program) without causing verification to fail. Uncovered specification elements indicate over-specification. Uncovered implementation elements indicate under-specification. Note that, in our interpretation, a subterm that appears identically more than once in a program may be covered in some locations and not others.

B. Interpretation

We begin by describing, at a high level, what program elements we consider atomic. This is of particular importance because the presence or absence of each type of element in the set of covered elements can mean something qualitatively different. Other choices are possible, including different choices in granularity, as we discuss further in Section III-E. But the following choices reflect the implementation in Boogie and Dafny that we have experimented with, and reflect an approach that would naturally apply to most verifiers for imperative languages, especially those that build verification conditions using the weakest precondition calculus.

In our formalization and implementation, we consider Boogie, a simple imperative language containing procedures and contracts on procedures. We include implementation statements and specification constructs as our atomic program elements. Implementation statements include assignments and calls to procedures. Specification constructs include assert statements, assume statements, loop invariants, and pre- and post-conditions on procedures.

One key type of information from coverage analysis is the identification of *uncovered* elements, and useful interpretation of coverage depends on the type of program element in the original language. For the constructs that show up in Boogie:

- An uncovered assignment or call statement indicates code that is either lacking specification (influencing no proof goals) or unreachable.
- An uncovered assume statement or precondition of the procedure being verified is a purely unnecessary assumption that can be removed.
- An uncovered precondition of a callee indicates that the precondition was established vacuously.

- An uncovered postcondition of a callee is an unnecessary assumption from the point of view of its caller. It may be necessary for other callers, but could be removed if it is not necessary for any caller.
- An uncovered postcondition, while verifying the procedure it is attached to, indicates that the postcondition is proved vacuously.
- An assert statement can be viewed as having two parts: an assertion that the expression is true at the point of the statement, and an assumption that it is true after the statement. If the former is uncovered, the assertion holds vacuously. If the latter is uncovered, the assertion is useful only for its top-level conclusion, not as an intermediate step toward proving later goals, and can be removed if it was intended for that purpose.
- A loop invariant that is uncovered while being proved indicates a vacuous proof.
- A loop invariant that is uncovered while assuming it in the body of the loop is an unnecessary invariant.

Note that provenance is critical for the approach to be useful, especially given that most verification tools work by successively lowering code into simpler and simpler languages. We'll show below that we can consider a *passive* [1], [12] language with only assert and assume statements, but we can provide much more useful feedback if we know what construct in the original language gave rise to each of these statements.

To make this precise, we now formally specify a language and how to compute verification conditions for it that enable coverage tracking.

C. VC Generation with Coverage Tracking

We first define verification condition generation with coverage tracking on the standard language of unstructured *passive* programs [1], [12], [18] shown on the top of Fig. 3. In a passive program, each assignment to a variable is represented by an assumption about a separate incarnation of that variable.

Note that the full Boogie language includes a larger set of constructs than this, and Section III-D describes how to track the necessary information for coverage analysis when translating those constructs into passive programs.

A command can be an assertion or an assumption of a predicate P annotated with a label L , and a block consists of a sequence of commands followed by a transfer to one or more successor blocks.¹ We extend this language slightly, however, by adding a label to each assertion and assumption. Coverage analysis is then the problem of identifying which statements, identified by label, are required to complete the proof.² This is fundamentally the information that unsatisfiable cores provide, when doing proofs with an SMT solver.

To construct verification conditions from labeled commands, we introduce a fresh Boolean variable, v_L for each label L , as

¹Note that this syntax doesn't include labels on blocks and syntactically ensures that the program is acyclic, a standard guarantee of passive programs.

²Previous work in ESC/Java labeled assertions for the opposite reason, to help identify what portions of a program participated in a counterexample [26]. The Simplify theorem prover [9] had built-in support for these labels.

c	$::=$	$\text{assert}_L P$
		$ \text{assume}_L P$
b	$::=$	$(c;)^* \text{goto } b^*$

$wp_C(\text{assert}_L P, Q)$	$=$	$(v_L \wedge P) \wedge Q$
$wp_C(\text{assume}_L P, Q)$	$=$	$(v_L \implies P) \implies Q$
$wp_C(\text{goto } b^*, Q)$	$=$	$\bigwedge_b wp_C(b, Q)$
$wp_C(c_1; c_2, Q)$	$=$	$wp_{c_1, wp(c_2, Q)}$

Fig. 3. Top: syntax of passive programs. Bottom: weakest preconditions of passive programs with coverage-tracking labels added. All instances of v_L are fresh.

shown on the bottom of Fig. 3. These variables interact with the postcondition in the same way that the predicate embedded within the command does.³

Once we have generated a verification condition C , we assert the entire negated VC, without a label, and assert each Boolean label along with a textual name. This leads to an SMT query like the following.

```
(assert v_L_1 :named "L_1")
...
(assert (not C))
(check-sat)
(get-unsat-core)
```

The SMT solver will respond to this query with a list of the labels that occur in the unsatisfiable core of the problem, corresponding to the covered program elements. Note that, in general, SMT solvers do not typically provide a *minimal* unsatisfiable core, and that multiple minimal cores may exist. This means that the possibility exists that a program element is included in the unsatisfiable core when, in fact, removing it may not lead to verification failure. Anecdotally, we have not seen lack of minimality cause issues in practice. However, an interesting avenue for future work would be to systematically measure how close cores are to minimal in practice, in this use case.

D. Desugaring with Labels

The Boogie language includes a wider range of commands than listed in the previous section, and already includes a transformation to desugar these into an acyclic graph of assertions and assumptions. This process is standard, and described elsewhere [1], [12], [18]. However, we provide an overview here of how it interacts with labels.

First, we define the syntax of labels used in Fig. 3 to include the alternatives shown in Fig. 4. In the surface Boogie syntax, an atomic label l can appear on an assignment statement, assert statement, assume statement, call statement, precondition, postcondition, or loop invariant. During desugaring, these labels are propagated as follows:

³The rule for assume statements already existed in Boogie, as a mechanism for tracking necessary assumptions as part of an effort to help better understand verification results. The portion of this focusing on errors has been published [6], but that publication doesn't discuss using the encoding to better understand successful proofs.

$L ::= l$	
PreCheck(l_1, l_2)	PreAssume(l_1, l_2)
Post(l_1, l_2)	InvEstablish(l)
InvMaintain(l)	InvAssume(l)

Fig. 4. Syntax of labels.

- Labels on assertions and assumptions are preserved.
- The label on an assignment is transferred directly to the corresponding assumption in the passive program.
- Each precondition (with label l_P) of a call (labeled l_C) is desugared into an assertion labeled with $\text{Pre}(l_C, l_P)$ and an assumption labeled with $\text{PreAssume}(l_C, l_P)$.
- Each postcondition (with label l_Q) of a call is translated to an assertion labeled with $\text{Post}(l_C, l_Q)$.
- Each invariant (with label l) is broken into three pieces: an assertion labeled with $\text{InvEstablish}(l)$, an assertion labeled with $\text{InvMaintain}(l)$, and an assumption labeled with $\text{InvAssume}(l)$.

The Dafny language includes even more constructs, and Dafny programs are verified by generating Boogie programs and verifying those using the standard Boogie verifier. The rules for tracking labels originating in Dafny language constructs are more numerous, but follow the same intuitive structure as the rules in the list above.

E. Finer Granularity

When calculating coverage using unsatisfiable cores, we can also consider Boolean sub-expressions of any larger expression. For example, if a precondition of the form $P \implies Q$ exists in the program, a coverage result that includes only P indicates that P is unsatisfiable and therefore the content of Q is irrelevant. The Boogie implementation includes decomposition of the Boolean operators for conjunction, disjunction, and implication.

Other implementations could go farther. As one extreme case, for bit vector programs, each bit could be considered separately. We have not yet experimented with this, but it would follow from the same approach. The finite nature of the programs would allow measurement of state space coverage, as well.

F. Impact of Trigger-Based Quantifier Instantiation

SMT solvers can use triggers [8], [9], [14] to aid quantifier instantiation. Each quantified formula may be annotated with a term schema that indicates the shape of a particular expression. If an expression of this shape exists in the current clause database, the solver may consider instantiating the quantified formula. In some cases, it is necessary to introduce a term into a Dafny program, potentially as an additional assertion, to allow the appropriate quantified axioms to be instantiated, even if that assertion is otherwise unnecessary for completing the proof. Such an element will be considered uncovered, even though removing it would cause verification to fail. This is

logically sound — the property to be proved is still derivable without that program element, even if doing it with SMT would require a perfect quantifier instantiation oracle in this case — but it can lead to a frustrating user experience in some cases.

IV. IMPLEMENTATION

To evaluate the precision, overhead, and usability of our approach on real-world verified programs, we implemented the analysis in two open-source tools: the foundation in Boogie, and extensions to Dafny to use this foundation.

The Boogie implementation integrates with the verification pipeline, building on the existing code for desugaring procedure calls and constructing an acyclic, passive program. The key changes needed were to allow statements to be annotated with labels and to extend labels during desugaring (as described in Section III-D). The implementation built on prior work intended to allow identification of necessary `assume` statements, making it also possible to identify covered assertions. This implementation consists of roughly 700 lines of C# code, almost all of which involves tracking the provenance of labels. The change to the VC generator exactly mirrors the description in Fig. 3.

Dafny verifies programs by first translating them into Boogie and then invoking the Boogie verifier. The Dafny implementation, consisting of around 2.5k lines of C# code, automatically adds a label to each Boogie statement that has a direct correspondence to a Dafny statement, and does additional provenance tracking to aid in producing useful messages. Based on its computation of coverage, Dafny implements warnings about contradictory or redundant specifications and source highlighting to indicate overall coverage and help identify unconstrained code.

A. Implementation Caveats

Building this analysis in Boogie and leveraging the existing Dafny to Boogie translation was mostly straightforward, but several subtle complications arose.

- Dafny programs sometimes include intentional proofs by contradiction. The goals of these proofs are typically uncovered, and therefore lead to warnings. To support these, we allow labeling an `assert` statement as taking part in an intentional proof by contradiction, silencing these warnings.
- Some of the Boogie code generated by Dafny includes assertions in locations that are unreachable by construction, presumably because it simplifies the translation and didn't previously lead to issues. We identify several classes of these and automatically exclude them from warnings.
- Each Dafny function definition is represented in Boogie as one or more quantified axioms, each of which includes most or all of the function body. This limits the degree to which we can track which subexpressions of a function body were covered by a particular proof. We can, however, track whether the function body was used at all. Inlining functions, rather than defining them with axioms, would

be one way to address this limitation, though it would not be possible for recursive functions.

V. EXPERIMENTS

Previous work on verification coverage has reported substantial overhead [16]. In addition, most tools for automatically detecting problems in software or specifications are prone to false positives, or a low level of “interesting” true positives. And, finally, SMT-based verification can be prone to the problem of *brittleness*. We say that a pair of an SMT solver and a query is brittle if the solver can solve the initial query but fails to solve a minor, semantics-preserving variant of the query.⁴ An analysis that increases brittleness too substantially may be infeasible to adopt in practice. Therefore, we aim to answer several research questions here.

RQ1 What is the typical overhead of calculating coverage, relative to doing verification without coverage analysis? We measure this in terms of both execution time and Z3 resource count (a deterministic measure of SMT solver work) per SMT query.

RQ2 Do the changes made to SMT queries to allow coverage analysis impact the brittleness of verification?

RQ3 How does the overhead compare to the “smoke testing” approach, where a tool attempts to prove *false* at various points throughout the program? In particular, how does it compare to the smoke testing implementation that already exists in Boogie?

RQ4 How useful are the warnings that Dafny generates? The possibility exists for entirely spurious warnings (due to encoding artifacts that lead to unreachable code or bugs in the implementation), but also for warnings about intentional proofs by contradiction or dead code, which may be considered uninteresting. The possibility also exists for false negatives, where program elements are determined to be covered due to the overapproximation of unsatisfiable cores.

To answer these questions, we ran Dafny on the code from several public repositories of Dafny code, and one non-public repository. These are:

- ESDK: AWS Encryption SDK for Dafny
- MPL: AWS Cryptographic Material Providers Library
- INT: A non-public project
- STD: Built-in Dafny `Std` library

We ran all measurements with a pre-release commit of Dafny (67daee75a302077b4b49048b4b90e560bfbd32b), with Boogie v3.0.12 and Z3 v4.12.1, on an AWS c5.4xlarge host (16 GHz Intel Xeon CPUs, 32GB RAM).

A. Results on Examples

Before presenting statistics about the performance of coverage analysis on large code bases, we show the results it provides on the examples shown in Section I. Recall that, in the first example (see Fig. 1), the `CallFind` method invokes

⁴Other authors have used the terms *instability* or the *butterfly effect* [32] for this phenomenon.

`Find` with arguments that make its postcondition unsatisfiable. Dafny reports this problem as follows:

```
Find.dfy(12,2): Warning: proved using
contradictory assumptions: assertion
always holds. (Use the '{:contradiction}'
attribute on the 'assert' statement to
silence.)
```

```
12 |   assert xs[i] == 45;
    |   ^^^^^
```

```
Find.dfy(12,9): Warning: proved using
contradictory assumptions: index in range
```

```
12 |   assert xs[i] == 45;
    |   ^
```

```
Dafny program verifier finished with 2
verified, 0 errors
```

Next, recall that, in the second example (see Fig. 2), the `BinSearch` method has an unnecessary precondition. Dafny reports this problem as follows:

```
BinarySearch.dfy(3,11): Warning: unnecessary
requires clause
```

```
3 |   requires forall i,j ::
    |           ^^^^^^^^^^^^^^^
```

```
Dafny program verifier finished with 2
verified, 0 errors
```

B. Overhead of Coverage Analysis

To determine how much overhead coverage analysis has relative to normal verification (RQ1), we measured the following (shown for each project in Fig. 5).

- the total number of goals for each project;
- time (in seconds) and resource count (RC, a deterministic proxy for difficulty provided by Z3) for each SMT query, for both normal verification (N) and verification with coverage tracking (C), along with statistics about the distribution of these values; and
- failure count, with and without coverage tracking.

In this data, we include the median, mean, and trimmed mean (a.k.a., truncated mean). This last value is calculated as the mean of the middle $n\%$ of values [7]. So the 98% trimmed mean (M98) discards the 1% smallest and 1% largest values. We calculate this because we observe that most Dafny verification tasks include a few very difficult queries and a huge number of relatively easy queries, and that the most difficult queries are frequently the most brittle. The trimmed mean discards the extreme outliers, giving a more characteristic picture of typical performance.

Based on these measurements, we compute an overhead, in terms of both time and resource count, for each project and for each statistic. These computations show that the overall time overhead ranges from around 20% to over 73%, depending

	ESDK	MPL	INT	STD
Goals	2122	4981	11042	4647
Time	N 7.8×10^2	7.2×10^2	4.9×10^3	2.8×10^2
	C 1.1×10^3	1.2×10^3	8.4×10^3	3.3×10^2
Mean	N 3.7×10^{-1}	1.4×10^{-1}	4.4×10^{-1}	6.0×10^{-2}
	C 5.1×10^{-1}	2.4×10^{-1}	7.6×10^{-1}	7.1×10^{-2}
	O 40.24%	65.66%	73.34%	19.97%
Med.	N 1.5×10^{-1}	5.2×10^{-2}	1.2×10^{-1}	2.4×10^{-2}
	C 1.5×10^{-1}	5.3×10^{-2}	1.1×10^{-1}	2.5×10^{-2}
	O 1.13%	2.51%	-4.27%	5.85%
M98	N 2.1×10^{-1}	1.2×10^{-1}	2.8×10^{-1}	5.3×10^{-2}
	C 2.1×10^{-1}	1.3×10^{-1}	3.0×10^{-1}	5.5×10^{-2}
	O 3.49%	9.17%	7.48%	4.37%
RC	N 9.4×10^8	2.2×10^9	4.7×10^9	8.0×10^8
	C 1.6×10^9	4.9×10^9	6.0×10^9	7.2×10^8
Mean	N 4.4×10^5	4.5×10^5	4.3×10^5	1.7×10^5
	C 7.7×10^5	9.9×10^5	5.5×10^5	1.5×10^5
	O 74.22%	121.37%	27.44%	-10.10%
Med.	N 2.7×10^5	8.7×10^4	1.0×10^5	5.4×10^4
	C 2.8×10^5	9.3×10^4	1.2×10^5	5.9×10^4
	O 3.33%	6.70%	14.18%	9.21%
M98	N 3.8×10^5	2.9×10^5	2.3×10^5	1.1×10^5
	C 4.2×10^5	3.4×10^5	2.8×10^5	1.2×10^5
	O 10.56%	17.59%	21.11%	6.77%
Fail.	N 2	1	3	0
	C 4	2	15	7

Fig. 5. Overhead of coverage-enabled verification with respect to normal verification. Each row is labeled with one of: “N” for normal verification, “C” for coverage-enabled verification, or “O” for calculation of the overhead of coverage analysis.

on the codebase. However, the median time overhead ranges between the much more tolerable -4% and 6% , and the trimmed mean between 3.5% and 9% . Similar patterns occur for resource counts. This suggests that, by constructing Dafny programs to avoid extreme outliers in verification time, we can keep the overhead of coverage analysis low enough to be almost imperceptible in practice. Our experience is that Dafny programs can be refactored to achieve this, and that doing so reduces verification brittleness in general.

We also show scatter plots comparing the resource count (Fig. 6) and time (Fig. 7) of each query between normal and coverage-enabled verification. Both plots include a random subsample of 1000 points from the total data set, and exclude a small number of outliers above 10^7 resource units or 3 seconds, to make the pattern of the remaining data clearer. These plots show that, although coverage-enabled SMT queries are usually more difficult, they can also sometimes be easier, and the vast

majority use less than twice the resources.

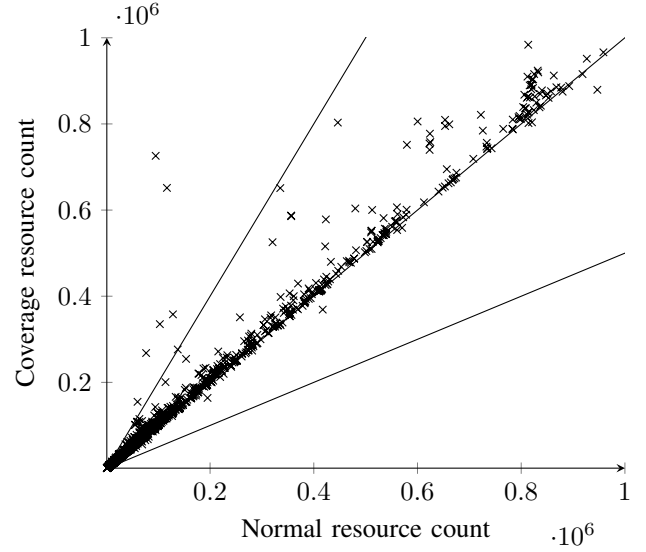


Fig. 6. Comparison of solver resource counts between normal (x-axis) and coverage-enabled (y-axis) verification, with lines for $y = 0.5x$, $y = x$, and $y = 2x$.

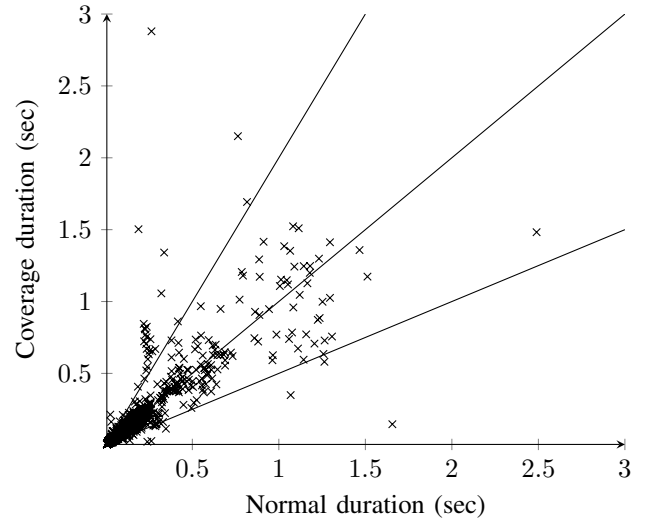


Fig. 7. Comparison of solver execution time between normal (x-axis) and coverage-enabled (y-axis) verification, with lines for $y = 0.5x$, $y = x$, and $y = 2x$.

C. Impact on Brittleness

To determine how much coverage analysis impacts the brittleness of verification (RQ2), we use a Dafny feature that allows each verification condition to be randomized along several dimensions: ordering of SMT declarations, names of SMT variables, and the random seed used by the SMT solver to make arbitrary decisions. The results of running 5 iterations of verification with this feature enabled, both with and without

coverage calculation, appear in Fig. 8.⁵ The increase in failure count and the standard deviation of resource count shows that enabling coverage analysis can increase brittleness — that is, it can increase the likelihood that verification will fail after small, semantics-preserving changes, and can increase the degree to which proof difficulty varies after such changes. However, the M98 trimmed mean shows only a slight increase in time and resource count, suggesting that the technique may be difficult to apply to already-brittle codebases, but is unlikely to introduce brittleness that is not already present.

	Mode	INT	STD
Time	N	1.4×10^4	1.4×10^3
	C	3.6×10^4	1.9×10^3
Mean	N	2.6×10^{-1}	6.1×10^{-2}
	C	6.5×10^{-1}	8.3×10^{-2}
Mean (98%)	N	1.3×10^{-1}	5.4×10^{-2}
	C	1.5×10^{-1}	5.7×10^{-2}
Std. dev.	N	5.2	2.6×10^{-1}
	C	1.2×10^1	1.3
RC	N	1.9×10^{10}	3.7×10^9
	C	3.5×10^{10}	5.0×10^9
Mean	N	3.5×10^5	1.6×10^5
	C	6.4×10^5	2.1×10^5
Mean (98%)	N	2.3×10^5	1.1×10^5
	C	2.8×10^5	1.2×10^5
Std. dev.	N	6.2×10^6	3.0×10^6
	C	1.0×10^7	6.9×10^6
Failures	N	18	13
	C	85	42

Fig. 8. Impact of coverage-enabled verification on brittleness.

D. Comparison with Smoke Testing

To compare coverage-enabled verification with smoke testing (RQ3), we focus on overall time and resource use, because smoke testing works entirely using additional queries, rather than modifying existing queries, and these therefore don't align with queries produced by normal or coverage-enabled verification. This also prevents a comparison in the form of a scatter plot. In addition, the implementation of smoke testing is less robust than that of coverage analysis, and fails on many of our benchmarks, so we could perform a meaningful comparison on only one element of our benchmark suite. The lack of robustness meant that it also needed to be run with slightly different Dafny flags, so the statistics in this table

⁵A bug in Dafny unrelated to coverage analysis prevented us from collecting brittleness data on the ESDK and MPL benchmarks. This bug has since been fixed, but not in time to be included in these results.

differ from those in Fig. 5. A summary of the comparison with smoke testing appears in Fig. 9.

	INT
Total coverage time	8.4×10^3
Combined smoke time	1.1×10^4
Total coverage resource count	6.0×10^9
Combined smoke resource count	2.1×10^{10}

Fig. 9. Overhead of smoke testing relative to normal verification, showing the total time and resource count taken by verification with coverage analysis enabled compared to the total time and resource count taken by verification with smoke testing enabled.

These results show that, in this instance, smoke testing is more expensive than coverage analysis. Other implementations of smoke testing could have different performance profiles. However, it is not clear that their performance could be directly compared to the implementation described in this paper, since they work on different languages.

Coverage analysis also provides additional capabilities, including detection of unnecessary assumptions, indication of what portions of an implementation have been verified, and indication of what facts were used in a proof.

E. Quality of Warnings

Finally, we analyze all of the warnings produced by coverage analysis and compare them with the warnings produced by smoke testing (RQ4). Coverage analysis warnings consist of redundant assumptions and contradictory assumptions. Redundant assumptions can be further categorized into:

- Redundant preconditions and assumption statements.
- Redundant assertions that were presumably added to aid in the proof of a later postcondition but turn out to be unnecessary.

The contradictory assumptions we encountered can be further categorized into:

- direct instances of `assume false`,
- dead code due to infeasible branches,
- inconsistencies in Dafny's background axioms,
- intentional proofs by contradiction,
- redundant tests of facts guaranteed by proof, and
- dead proof or specification code due to infeasible cases.

The number of each of these warnings appear in Fig. 10. Direct instances of `assume false` are a mechanism for avoiding proof of difficult properties, and occur in two of the projects analyzed. Dead code is rare, occurring in only two instances, in one project. The process of performing these experiments uncovered an inconsistency in the background axioms Dafny generates related to the freshness of heap objects that affects one project (and has since been fixed). Three of the four projects include intentional proofs by contradiction. Two of the projects included redundant runtime tests of properties that were guaranteed by proof. And three of the projects included either specification expressions in which some subexpression was constant or branches in proofs that covered impossible

	ESDK	MPL	INT	STD
Coverage warn.	38	301	78	290
Redund. assum.	24	164	23	59
Redund. assert.	11	30	25	201
False assum.	0	0	8	9
Dead code	0	2	0	0
Prelude contrad.	0	69	0	0
Proof by contrad.	2	15	0	14
Redundant test	1	8	0	0
Dead proof/spec.	0	13	20	7
Smoke warn.	-	-	2897	-

Fig. 10. Warnings produced from coverage-enabled verification and smoke testing.

cases. These could be removed, simplifying the specification and proofs.

The inconsistencies due to Dafny’s prelude axioms were the most significant specification mistake detected in this analysis. However, all of the others, except the intentional proofs by contradiction, indicate implementation or specification code that could be improved.

Smoke testing generates warnings in terms of Boogie code rather than Dafny code. That, plus the large number of warnings, made it infeasible to categorize each one. We instead simply note that the sheer number, even if they were in Dafny terms, would make them infeasible for the developers to triage, as well.

VI. CONCLUSION

The concept of verification coverage originated in the model checking world, where the monolithic nature of many model checking systems meant that coverage analysis was often expensive. This analysis has a direct analog in compositional deductive verification of software, however, where individual proof goals tend to be smaller and simpler. We have shown that coverage analysis can be smoothly integrated into a verification-aware programming language like Dafny with reasonable performance overhead and problem reports free of false positives (with caveats for partially unused elements and intentional proofs by contradiction), though false negatives (failures to detect true specification gaps) are possible.

This work only begins the investigation into coverage in deductive software verification. Several avenues for future work are immediately apparent:

- Coverage can be calculated with finer granularity (see Section III-E). Labeling additional program elements may make it possible to provide more precise feedback to the programmer about the quality and scope of their specifications.
- Unsatisfiable cores can be minimized (with additional computational expense). This has the potential to reduce or eliminate the possibility of false negatives, although multiple minimal cores may exist in some cases. Alternatively,

analyzing the structure of the proof of unsatisfiability, rather than the clauses in the unsatisfiable core, also has the potential to produce more precise results [28].

- Information about the facts used in a proof can be used to optimize future searches for the same or similar proofs. A Dafny lemma, for example, could be updated to restrict the facts sent to the SMT solver to include only those that were necessary to complete the proof, reducing the solver’s search space.

REFERENCES

- [1] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, September 2005.
- [2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021.
- [3] Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. Towards a notion of coverage for incomplete program-correctness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 53–63, Cham, 2018. Springer International Publishing.
- [4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. <https://hal.inria.fr/hal-00790310>.
- [5] Boogie. <https://github.com/boogie-org/boogie>.
- [6] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholtz. Integrated environment for diagnosing verification errors. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 424–441, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
- [7] Anirban DasGupta. *Asymptotic Theory of Statistics and Probability*, chapter 18. Springer Texts in Statistics. Springer-Verlag New York, 2008.
- [8] 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, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [9] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
- [10] William R. Elmendorf. Controlling the functional testing of an operating system. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):284–290, 1969.
- [11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, October 2001.
- [12] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’01, page 193–205, New York, NY, USA, 2001. Association for Computing Machinery.
- [13] Frama-C. <https://www.frama-c.com/>.
- [14] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 167–182, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [15] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 314–325, New York, NY, USA, 2016. Association for Computing Machinery.
- [16] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats P. E. Heimdahl, and Lucas Wagner. Proof-based coverage metrics for formal verification. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 194–199, 2017.
- [17] Mihaela Gheorghiu and Arie Gurfinkel. VaqUoT: A tool for vacuity detection. Technical report, University of Toronto, Department of Computer Science, 2006.

- [18] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, New York, NY, USA, 2009. Association for Computing Machinery.
- [19] Jochen Hoenicke, K. Rustan M. Leino, Andreas Podelski, Martin Schäfer, and Thomas Wies. It's doomed; we can prove it. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 338–353, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [20] Mikoláš Janota, Radu Grigore, and Michal Moskal. Reachability analysis for annotated code. In *Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, page 23–30, New York, NY, USA, 2007. Association for Computing Machinery.
- [21] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. On the duality between vacuity and coverage. Technical Report UCB/EECS-2008-26, EECS Department, University of California, Berkeley, March 2008.
- [22] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08. IEEE Press, 2008.
- [23] Viet Hoang Le, Loïc Correnson, Julien Signoles, and Virginie Wiels. Verification coverage for combining test and proof. In Catherine Dubois and Burkhart Wolff, editors, *Tests and Proofs*, pages 120–138, Cham, 2018. Springer International Publishing.
- [24] K. Rustan M. Leino. This is Boogie 2. June 2008.
- [25] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [26] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1–3):209–226, March 2005.
- [27] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [28] Jocelyn Simmonds, Jessica Davies, Arie Gurfinkel, and Marsha Chechik. Exploiting resolution proofs to speed up ltl vacuity detection for bmc. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 3–12, 2007.
- [29] Aaron Tomb and Cormac Flanagan. Detecting inconsistencies via universal reachability analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 287–297, New York, NY, USA, 2012. Association for Computing Machinery.
- [30] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, pages 326–341, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [31] Why3 – where programs meet provers. <https://www.why3.org/>.
- [32] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 178–188, 2023.

APPENDIX

A. Experimental Reproduction

This appendix describes how we gathered experimental data for each benchmark listed in the paper. Each of the following subsections describes one benchmark, giving the URL of the repository on GitHub, the SHA of the commit we ran on, the subdirectory of the repository we ran the verification in (if not the root), and the verification script we ran. We used different verification scripts to mirror each project's standard build setup as closely as possible.

1) *Common Options*: Each of these scripts passes its arguments to Dafny. This is so that we can use the same verification scripts for each mode of analysis we performed.

For each experiment type, we passed in old-style Dafny CLI flags, because that is what each project used in its build system at the time of writing. These arguments are as follows:

For all experiments:

- /verificationLogger:csv

For coverage:

- /warnRedundantAssumptions
- /warnContradictoryAssumptions

For brittleness analysis:

- /randomizeVcIterations:5

For smoke testing:

- /smoke
- /trace
- /vcsCores:1
- /prune:0
- /restartProver

2) *STD*:

- Repository URL:
 - <https://github.com/dafny-lang/dafny>
- Commit hash:
 - 67daee75a302077b4b49048b4b90e560bfbdf32b
- Subdirectory:
 - Source/DafnyStandardLibraries/src/Std
- Verification script:

```
dafny \
  /compile:0 \
  /trackPrintEffects:1 \
  /definiteAssignment:3 \
  /readsClausesOnMethods:1 \
  /rlimit:1000000 \
  `find . -name "*.dfy" | \
    grep -v "TargetSpecific" ` $*
```

3) *MPL*:

- Repository URL:
 - <https://github.com/aws/aws-cryptographic-material-providers-library>
- Commit hash:
 - 29c6a2c20d4cdeaadeacd242dc4764ce0012193d
- Verification script:

```
find . -name "*.dfy" | \
  grep -v "libraries/" | \
  xargs -P 4 -n 1 \
  dafny \
    /compile:0 \
    /definiteAssignment:3 \
    /quantifierSyntax:3 \
    /functionSyntax:3 \
    /unicodeChar:0 \
    /timeLimit:150 $*
```

4) *ESDK*:

- Repository URL:
 - <https://github.com/aws/aws-encryption-sdk-dafny>
- Commit hash:
 - 24ddf9a7b2840bea4a37c633c45d60cdfc177184
- Verification script:

```
find . -name "*.dfy" | \
  grep -v "libraries/" | \
  grep -v "mpl/" | \
```

```
grep -v "test/" | \
xargs -P 4 -n 1 \
  dafny \
    /compile:0 \
    /definiteAssignment:3 \
    /quantifierSyntax:3 \
    /functionSyntax:3 \
    /unicodeChar:0 \
    /timeLimit:150 $*
```

A Tale of Two Case Studies: A Unified Exploration of Rust Verification with SEABMC

Joseph Tafese*
University of Waterloo
Waterloo, Canada
jetafese@uwaterloo.ca

Siddharth Priya*
University of Waterloo
Waterloo, Canada
siddharth.priya@uwaterloo.ca

Giuliano Losa
Stellar Development Foundation
San Francisco, USA
giuliano@stellar.org

Arie Gurfinkel
University of Waterloo
Waterloo, Canada
agurfink@uwaterloo.ca

Graydon Hoare
Stellar Development Foundation
San Francisco, USA
graydon@stellar.org



Abstract—The Rust type system provides strong compile-time guarantees. However, some properties cannot be fully verified by the compiler. Specifically, properties like panic freedom and memory safety in mixed safe-unsafe code require verification beyond what the language enforces. We explore how to verify these properties in real-world Rust code using SEABMC, a bounded model checker that ingests LLVM-IR generated by the Rust compiler. We demonstrate our approach through two case studies. In the first, we develop unit proofs for functional properties of four data-structure libraries: SMALLVEC, TINYVEC, SEAVEC, and RESULT-TYPE from the Rust standard library. These unit proofs are checkable by both SEABMC and KANI, a state-of-the-art bounded model checker for Rust, and we find that SeaBMC verifies these units an order of magnitude faster than Kani. The second case study focuses on verifying panic freedom of Wasmtime’s WINCH compiler. This application is driven by the requirement for high reliability when compiling WASM smart-contracts in the Stellar network. This case study highlights that executable counterexamples from SEABMC are highly effective for localizing issues and discovering invariants. Our main contributions are (1) a new tool for Rust verification which, on our benchmarks, is an order of magnitude faster than KANI, (2) two case studies with reusable benchmarking and testing infrastructure, and (3) practical guidelines stemming from our experience verifying real-world code-bases.

I. INTRODUCTION

The Rust type system makes progress on long standing safety problems in systems programming—most notably, it ensures the absence of memory-safety violations, data races, and other undefined behavior. However, the Rust type system guarantees those safety properties only for a subset of the language called Safe Rust. For expressiveness and speed, programmers must often resort to Unsafe Rust. Thus, the Rust type checker does not guarantee the safety of many practical Rust programs.

Verification tools that work on mixed safe-unsafe Rust code can fill an important gap left by the Rust type system. In practice, undefined behavior stemming from unsafe Rust code may lead to panics (an exception denoting an unrecoverable error), crashes, or subtle memory-corruption issues. We treat

```
fn main() {
    let arr = [10, 20, 30, 40, 50];
    let p = arr.as_ptr();
    // Move pointer FAR past the array bounds
    let bad_ptr = unsafe { p.add(10) };
    let bad_index = unsafe { bad_ptr.offset_from(p) } as usize;
    println!("Value: {}", arr[bad_index]);
}
```

Fig. 1: Verifying panic freedom in the presence of unsafe code requires low-level reasoning.

panics and undefined behavior interchangeably, unless specifically mentioned. For example, in Fig. 1, an unsafe Rust block creates an invalid pointer at line 5, which leads to a runtime panic at line 7. This style of programming adds security defensively. However, a program may wish to ensure no panics occur, i.e., panic freedom. Thus, panic freedom becomes an important property to prove.

In this paper, we present a verification methodology for mixed safe-unsafe Rust code based on the SEABMC bounded-model checker. SEABMC is part of the SEAHORN verification toolkit. It analyzes LLVM bitcode produced by the Rust compiler and massaged by SEAHORN’s pre-processing passes, and relies on an SMT solver to discharge verification conditions. Since SEABMC works on the low-level representation of the program post Rust compilation to LLVM, it can verify mixed safe-unsafe Rust, and even mixed Rust and C programs.

A SEABMC user must assemble three components for verification. The first is the system under test (SUT). This may be a single function or a more complex arrangement of system components. The second is the runtime environment needed by the SUT. This may be part of the language’s runtime system (memory allocation, error handlers, etc.), which we are not interested in verifying, but it may also contain system components that the SUT uses but that we are not verifying. During verification, the runtime environment is replaced by a simpler verification-only environment. Third, a unit proof [1] (similar to a unit test) that sets up concrete and symbolic inputs and state needed for the SUT to execute. It usually has three stages—setup pre-conditions, call SUT, check post-

* Authors made an equal contribution.

conditions. We apply this methodology in two case studies.

In the first case study, we write 83 unit proofs that verify low-level functional correctness. We compare SEAHORN and KANI [2] on these unit-proofs. KANI is a bounded model checker for mixed safe-unsafe Rust code developed by AWS and based on CBMC [3], and it has recently seen a flurry of activity [4]. Both tools are configured to automatically check spatial memory safety. We show that SEABMC is faster than KANI by a factor of 10 or more under similar checks. Additionally, we provide a new design point around panics in unit proofs. KANI adopts the view that any panic is always an error (PANIC-ERROR). However, this can force users to write unit proofs with very strong pre-conditions in order to avoid *any* panic. An alternative is to abort execution upon a panic, without considering it an error (PANIC-ABORT). The PANIC-ABORT approach allows writing weaker pre-conditions quickly. The executions that panic simply exit early. This style of unit proofs is inspired by smart contracts written in Rust that use panics to safely terminate an incorrect execution early. In summary, SEABMC can meet the challenge of verifying low-level safe-unsafe Rust code by being precise and fast.

In the second case study, we deploy our methodology to verify parts of WINCH, an industrial WASM-to-executable compiler for various architectures. With the aim of identifying invariants that ensure panic freedom, we contribute 14 unit proofs that check properties of a system under test (SUT) consisting of roughly 3,300 lines of Rust code (LOC). We use SEABMC to verify panic freedom of core compiler components since the usage context is especially sensitive to nondeterministic panics. During this exercise, we discover a panic that encodes a requirement: WINCH must run on an architecture with 64-bit pointers. Without assuming this precondition, SEABMC reaches a panic, and SEAHORN provides an executable counter example that makes it easy to discover the 64-bit pointer-size requirement.

We show that SEABMC meets the challenge of verifying Rust programs which mix both safe and unsafe code. The SEABMC tool and case studies are open source. The rest of the paper is organised as follows. In Section II, we describe how SEABMC works on LLVM bitcode. Section III describes how unit proofs for Rust are setup. In Section IV we describe how SEABMC compares to the state-of-art KANI tool. We provide our experience verifying parts of the WINCH compiler using SEABMC in Section V. Future and related work is discussed in Sections VI and VII, respectively. We conclude in Section VIII. Artifacts for both the VERIFY-RUST and WINCH case studies are available online¹.

II. ENGINEERING SEABMC FOR RUST

SEAHORN is a verification toolkit for LLVM programs. SEABMC is a bit-precise bounded model checking engine for SEAHORN that has been used in various studies ([1], [5], [6]). LLVM IR produced by a language frontend is transformed by the SEAHORN pre-processing pipeline into a representation

where memory dependencies between LLVM operations are made explicit. This intermediate representation is called SEA-IR [7] and is the input for SEABMC. SEABMC was originally developed for verifying LLVM IR produced by the CLANG compiler for C. It works by symbolically unrolling a program's control flow-loops, function calls, and conditional branches, up to a user-specified bound. It then generates verification conditions that capture all reachable states within that bound. These conditions are translated into SMT formulas and passed to a solver (Z3 [8] by default) which checks whether they are satisfiable, which would indicate that the program can reach an error state, e.g., a panic or an invalid memory access. When SEAHORN determines that an error state is reachable, it generates a harness that drives the SUT to the error state; after compiling and linking this harness with the SUT, the user can diagnose the issue with a standard debugger. For example, if a 32-bit symbolic value must be zero to trigger the panic, SEAHORN generates a harness that provides a 32-bit zero value. We will discuss this with an illustrative example in Section V-B.

For memory safety, SEABMC uses fat-pointers to carry allocation start and end addresses (bound) apart from pointed-to address. We also use shadow-memory to encode invariants such as allocation bounds, memory read/write, and no use-after-free. Moreover, the SEABMC memory allocator guarantees that fresh memory allocations are disjoint in the address-space from existing allocations. With this, pointer provenance reduces to checking if a fat pointer is within its allocation bound. More details can be found in [7].

Using SeaBMC with Rust typically involves compiling the code to a supported LLVM version (e.g., Rust v1.64 for LLVM 14). No modification of SeaBMC itself is required, since the tool treats LLVM IR uniformly regardless of whether it originated from C or Rust. Note, our experience is different from [9] where some previously unhandled LLVM instructions had to be handled for Rust. In our case, all relevant LLVM instructions generated by the Rust compiler are handled by SEABMC out-of-box. This compatibility illustrates a broader point: verification tools built for C can often be applied to Rust with little effort, provided they operate at the LLVM-IR level. However, some engineering work is needed to massage Rust compiler LLVM output to the right form for verification.

First, we have to simplify the Rust startup code. This code is responsible for setting up the runtime environment, and it executes before the Rust unit proof is called. The startup logic can be complex to execute symbolically. Moreover, the complex startup code is not needed for verifying the unit proof since SEABMC provides simpler stub implementations for this functionality. To remove this startup code, one option is to remove the target functions from the LLVM bitcode generated by the Rust compiler. This method used in [10] has limited success since a list of candidate functions to remove has to be maintained across Rust versions. Instead, we use a novel mechanism that treats the rust code as a library and links it to a C function that serves as the entry point for verification. During linking, Rust does not embed the unneeded functions

¹<https://doi.org/10.5281/zenodo.16415667>

```

#[cfg_attr(kani, kani::proof)]
#[cfg_attr(kani, kani::unwind(5))]
#[cfg_attr(kani, kani::should_panic)]
#[no_mangle]
fn test_push() {
  const CAP : usize = 4;
  let mut v: ArrayVec<u32; CAP> = ArrayVec::new();
  let len: usize = verifier::any!();
  verifier::assume!(len <= CAP);

  for i in 0..len {
    v.push(verifier::any!());
    verifier::vassert!(v.len() == i + 1);
  }

  verifier::vassert!(v.capacity() == CAP);

  if len == CAP {
    // Vector is at capacity, so push should panic.
    v.push(verifier::any!());

    // This assertion should not be reachable
    // since the previous push panics.
    verifier::error!();
  }
}

```

Fig. 2: Unit proof for the TINYVEC push operation.

since the C runtime is designated active. This works well because SEAHORN already provides a runtime environment for a C executable ².

Second, getting access to a LLVM bitcode representation of the complete Rust program is not straightforward. By default, the Rust compiler does not provide information on all the intermediate LLVM bitcode files from different compile units before linking. For example, the `emit-llvmir` option does not provide definitions of the allocation functions from the standard library. Without these definitions, the obtained LLVM code is incomplete and would require SEAHORN to patch the LLVM bitcode with missing definitions. This patching is not a clean or easily maintainable solution. Thus, we must obtain a complete LLVM program. For this, we rely on LLVM’s FatLTO [11] feature. FatLTO produces a native binary that additionally embeds a complete LLVM program, which we then extract and input to SEAHORN.

III. VERIFYING RUST

We want SEABMC to be a testbed for exploring new ideas in verification of Rust programs. Towards this goal, we have created a benchmarking suite of Rust unit proofs with three goals in mind – portability, comparison to state-of-the-art, and, extensible experiments.

Unit proofs and CAS. Recall that a unit proof sets up concrete and symbolic inputs, pre-conditions, calls the system under verification (SUT), and checks the post-conditions. To make writing pre-and-post conditions natural to developers, it is useful to use code as specification (abbrev. CAS) [12]. We want unit proofs in VERIFY-RUST to be tool agnostic. For this, we have developed VERIFIER-LIB, a Rust crate that provides a single API to write SEABMC and KANI specifications as assumptions and assertions in the unit proof directly.

An example unit proof for the TINYVEC push operation, shown in Fig. 2, illustrates these concepts. Lines 1–4, are

²The WINCH case study uses a superficially different setup, but this is a historical accident.

configuration options for KANI. The first line marks the function as a KANI proof. The second line sets the loop unwinding bound to 5. The third line marks that the code is expected to panic. KANI treats all panics as verification failures otherwise. The `#[no_mangle]` attribute tells the Rust compiler to not mangle [13] the function name. It is required because SEAHORN verification infrastructure uses the LLVM function as a key to run and display test results. Going to the unit proof itself, lines 6–9 setup the unit proof preconditions. The vector capacity is set to four elements. The length is designated as a non deterministic value using `verifier::any!()`. It is then constrained to between zero and capacity inclusive. Note that the `verifier::assume!` and `verifier::vassert!` macros are from VERIFIER-LIB. The specification is delegated to corresponding SEABMC or KANI API when the unit proof is compiled. Lines 11–14 call the SUT in a loop. After each call, the unit proof checks that the vector length has indeed increased by one. Finally line 16 checks that the capacity is unchanged. Now, an additionally property of TINYVEC push is that it should panic if the vector is already at capacity. This is checked in lines 18–24. Here, the unit proof goes to an error state if the push did not panic.

Panic behaviour. SEABMC provides control on how the unit proof is to behave when a panic is encountered. One mode treats a panic as an error (PANIC-ERROR). Another mode aborts execution on panic without error (PANIC-ABORT). For SEABMC, these modes are configured separately from a unit proof by configuring the Rust compiler with different panic handlers. Thus, the same unit proof can be run in different modes without modifying the proof code itself.

For example, in Fig. 2, we can remove line 9. Now when the proof is executed in PANIC-ABORT mode (with sufficiently large bound), some program executions will encounter a panic in push. This is because TINYVEC cannot grow beyond capacity. These executions will silently abort. The resulting verification conditions of the program may be easier to solve since panics are considered unreachable. Tool performance in PANIC-ABORT mode is discussed more in Section IV. Note that PANIC-ABORT mode also allows a proof design pattern to expect that a specific computation always panics. This is done in the form of an unreachable `verifier::error!()` as in line 24. Capturing that a specific panic should occur in KANI is not possible as it reports *any* panic in the unit proof as an error. There is no PANIC-ABORT mode in KANI.

IV. BENCHMARKING

To examine how SEABMC works for practical Rust programs, we wrote unit proofs for four Rust libraries – TINYVEC, SMALLVEC, SEAVEC, and, RESULT-TYPE. The unit proofs cover common operations on data structures. The TINYVEC library provides a vector like data structure backed by an array. It panics if elements are added to it beyond capacity. The SMALLVEC library uses the stack for small vector sizes. Beyond this small size, SMALLVEC uses the system heap to store vectors. Many SMALLVEC internal operations are marked unsafe. The SEAVEC library is a minimal vector library to

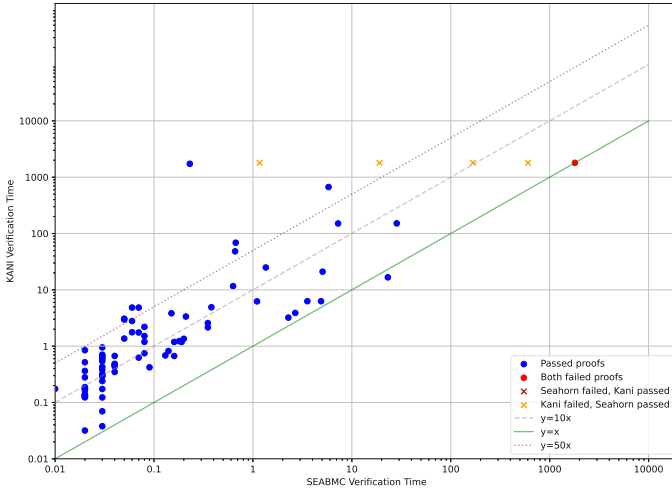


Fig. 3: Verification time (in sec.) using KANI vs SEABMC (using PANIC-ERROR).

serve as environment for verification of programs that require a vector implementation. In SEAVEC, memory is allocated on the heap and uses unsafe code. However, unlike SMALLVEC, there is no way to grow the vector beyond capacity. The `RESULT-TYPE` is a type used for returning and propagating errors in the Rust standard library. We have written 38 proofs for TINYVEC, 29 proofs for SMALLVEC, 9 for SEAVEC, and, 7 for `RESULT-TYPE`, for a total of 83 proofs. These proofs verify properties of common operations on the data structures. For example, for list data structures, the unit proof may check that a push increases the vector length by one and that push pushing to a list at capacity causes a panic. For `RESULT-TYPE`, the proof may check that a reference inside an `std::result` enum refers to the expected value. All proofs are checked for spatial memory safety by both SEABMC and KANI.

Setup. One of the goals of the benchmark data set is to be reproducible, debuggable, and extensible. For this we use the ReFrame [14] testing framework. ReFrame is mature, widely supported, and offers convenient test configuration syntax based on Python3. Some of its features we found useful were (1) the in-built design to separate test policy (what to test) from mechanism (how to test), (2) convenient syntax to search for strings in test output and operate on them – e.g., sum of all timings in `stdout`, and (3) staging for each test such that failures are reproducible for debugging

SEABMC and KANI. Figure 3 graphs verification time of KANI vs SEABMC as a scatter-plot on TINYVEC, SMALLVEC, SEAVEC, and, `RESULT-TYPE`. Both the axes are in log scale. The timeout for both KANI and SEABMC is set to 30 minutes. KANI times out for five unit proofs. SEABMC times out for one unit proof. We find that SEABMC is usually an order of magnitude faster than KANI.

We note that an apples-to-apples comparison of verification time between SEABMC and KANI is hard. We do not consider wall time because the compilation pipeline of the two tools differ. KANI plugs into the Rust compiler as a backend code generator. SEAHORN is invoked only after code generation to LLVM is complete. To compare time spent in bounded model

```
#[cfg_attr(kani, kani::proof)]
#[cfg_attr(kani, kani::unwind(3))]
fn test_simplify_cfg() {
    let v: u8 = verifier::any!();
    verifier::assume!(v < 3);

    for i in 0..v as usize {
        return;
    }

    let mut sentinel: u32 = verifier::any!();
    // INV: v == 0 => sentinel >= 0
    verifier::vassert!(sentinel >= (v as u32)*(v as u32));
}
```

Fig. 4: LLVM optimization discharges all asserts in above program before SEABMC is invoked.

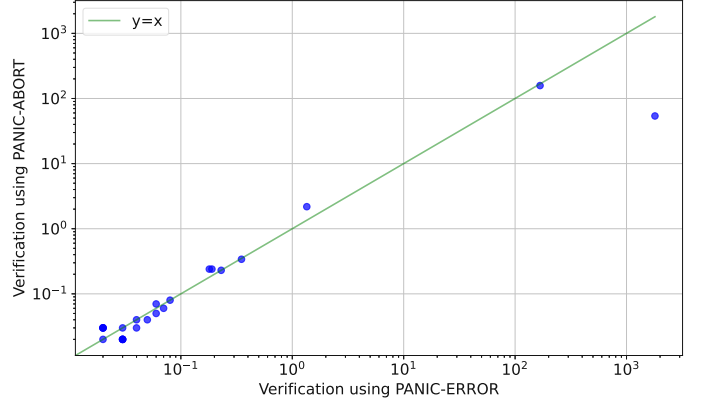


Fig. 5: Time (in sec.) to verify using SEABMC in PANIC-ABORT mode vs PANIC-ERROR mode.

checking, we consider verification time of KANI as logged by it after a series of post processing steps. The verification time of SEABMC is the tool running time after preprocessing.

While SEABMC shows consistently better performance than KANI on our benchmarks, there may be no single explanation for this. Both tools are multi-staged with unique strategies for generating verification conditions. Thus, it is difficult to *diff* where the added performance comes from. For SEABMC, we have found that the LLVM optimization passes that run as part of Rust compilation and during SEAHORN preprocessing can statically simplify the code considerably. We illustrate using the unit proof in Fig. 4. An execution may enter the loop in line 7 and exit early. Or, it may not enter the loop ($v = 0$). Thus, the only way to reach line 11 is when $v = 0$. Further, the variable `sentinel` is an unsigned 32-bit number. With these two facts, the `vassert` in line 13 becomes trivially true since the type of `sentinel` ensures that it is always greater than equal to zero. This reasoning is done statically in the Rust + SEAHORN pipeline and the `vassert` is discharged before invoking VC generation. In our experiments, KANI invokes the solver indicating that it relies on the solver to reason as above. Invoking the solver is usually expensive, therefore, any pre-solver reasoning should improve verification time.

PANIC-ABORT vs PANIC-ERROR. In order to scale SEABMC to verify more complex code, we benchmark 24 unit proofs in TINYVEC to run in PANIC-ABORT vs PANIC-ERROR mode. The results are shown in Fig. 5. On average, we see that PANIC-ABORT is 2.4x faster than PANIC-ERROR.

This is somewhat skewed by `test_retain` that times out with SEAHORN (30 minutes) in PANIC-ERROR mode and takes 54 seconds in PANIC-ABORT mode. As discussed in Section III, this speedup occurs because PANIC-ABORT leads to simpler unit proofs since weaker assumptions can be made.

V. APPLYING SEABMC TO WINCH

In this section, we present a case study in which we use SEAHORN (and SEABMC) to verify a real-world codebase: the WINCH compiler. WINCH compiles WebAssembly (WASM) code to native binaries. It is part of Wasmtime [15], an open-source project maintained by the Bytecode Alliance. We undertake its verification to support high-reliability compilation of WASM smart contracts³ in the Stellar network [16], a popular blockchain.

Compared to the benchmarks of Section IV, we face several new challenges. First, the codebase is too large and complex to create unit proofs that exercise the end-to-end compilation of even small WASM programs. We, therefore, need to carve out a manageable SUT. This involves writing stubs for internal WINCH components and creating mixed concrete/symbolic contexts to exercise the SUT. Performing this surgery requires a good understanding of the internal structure of WINCH. Second, components that form the environment of the SUT likely maintain state invariants that the SUT depends on; if the context we create for the SUT does not satisfy those invariants, we get false-positive verification failures. Third, the WINCH codebase uses data-structures such as vectors with dynamic resizing, hash-maps, and hash-sets that are difficult for SEAHORN to execute symbolically. To make analysis feasible, we must replace these data-structures by stubs that are more tractable while exhibiting sufficiently rich behavior. Finally, to deliver long-term value, we need to be able to maintain the SUT and keep it up to date with changes to the upstream codebase.

In the next two sections, we describe these challenges in more detail, offer guidelines born from experience, and show that SEAHORN can realistically be applied to real-world code bases. Before we jump in, let us say a few more words about our motivations.

The Stellar network implements a fault-tolerant distributed ledger using state-machine replication[16]. Hundreds of replicas across the world each maintain a copy of the full ledger state, and users can access the ledger by invoking pieces of WASM code called smart contracts. Smart contracts are themselves user-provided, and users can register new smart contracts at any time. To keep their state synchronized, the replicas use a consensus protocol to repeatedly agree on the next operation to execute, e.g. adding a new smart contract to the system, or invoking an existing smart contract.

Currently, the replicas execute WASM code using the Wasmi interpreter [17]. However, in a proposed upgrade, replicas instead use WINCH to compile WASM code to native

binaries that are then cached for later execution. Since users rely on the Stellar network for potentially high-value financial transactions, it is crucial that WINCH not crash, panic, or miscompile WASM code. Moreover, the state-machine replication protocol used in the Stellar network relies on identical, deterministic replica behavior. It is thus also important to verify that the behavior of WINCH does not depend on environment parameters (i.e. instruction set, operation system) that may vary across replicas.

For this case-study, we focus on verifying panic-freedom of WINCH’s x86_64 target. We therefore employ the PANIC-ERROR approach to panics, as described in Section III.

A. Carving out the System under Test

WINCH is a single-pass compiler that compiles WASM to native machine code. Fig. 6 illustrates at a high level the architecture of WINCH. The parser parses the input WASM code and the validator checks that it is valid, as defined by the WASM specification [18]. The result is an abstract syntax tree (AST) that has been validated. WINCH then traverses this AST in a single pass using the visitor design pattern [19]. For each AST node, a visit function corresponding to the node type starts a processing sequence involving the CODEGEN CONTEXT (responsible for high-level orchestration), MACRO ASSEMBLER (responsible for architecture-specific concerns) and ASSEMBLER (responsible for binary encodings), culminating with instructions being emitted to the MACHINE BUFFER.

WINCH borrows the parser and validator components from `wasmparser`⁴ [20], and the MACHINE BUFFER component from the Cranelift compiler backend (also developed by the Wasmtime project). We focus our verification efforts on the WINCH-specific components exclusively. These components have not been battle-tested in other projects and may therefore harbor more issues. The SUT thus consists of the VISITOR’s visit functions and the CODEGEN CONTEXT, MACRO ASSEMBLER, and ASSEMBLER components.

To prepare the SUT, we first modify WINCH to compile with Rust 1.64 in order to obtain LLVM 14 bitcode, the latest version currently supported by SEAHORN. To satisfy this requirement, we prune or downgrade several dependencies. Second, save for a few necessary type definitions, we remove the parser and the validator, and stub the MACHINE BUFFER using a minimal implementation simulating expected behavior. This allows us to isolate WINCH-specific components and obtain a tractable yet non-trivial SUT (about 3,300 lines).

Finally, we create a proof environment that sets up the necessary context for calling the SUT. Our entry points are the VISITOR’s visit functions, which we provide with a mix of symbolic and concrete inputs. We also initialize the state of necessary WINCH components (e.g. the register allocator, stack frame, etc.) and set up various data structures. In all cases, we may use both concrete and symbolic data. To avoid spurious panics, we must constrain initial state and inputs to

³A program that executes predefined actions automatically when specific conditions are met.

⁴An open-source project maintained by the Bytecode Alliance.

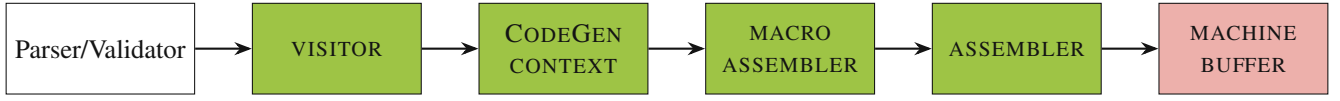


Fig. 6: WINCH Architecture showing, in order, trusted, verified and stubbed components.

```

1  #[no_mangle]
2  fn visit_cmp_ops() {
3    let vmoffsets = VMOffsets::new();
4    let codegen_context = proof_core::setup_context(&vmoffsets);
5    let mut emission_context = codegen_context.for_emission();
6    let mut masm = proof_core::setup_masm();
7    // SUT
8    // invariant: second value on stack should be dst reg
9    let dst = Reg(PReg::new(2, regalloc2::RegClass::Int));
10   emission_context.stack.push(Val::Reg(TypedReg::i32(dst)));
11   let kind = IntCmpKind::from(nondet_u8());
12   // call functions by operation width
13   let v2 = nondet_u8();
14   let res = match v2 {
15     0 => {
16       // invariant: need immediate/register at stack top
17       let val = nondet_i32();
18       emission_context.stack.push(Val::I32(val));
19       emission_context.i32_binop(&mut masm, |masm, dst, src, size| {
20         masm.cmp_with_set(writable!(dst), src, kind, size)?;
21         Ok(TypedReg::i32(dst))
22       })
23     },
24     _ => {
25       // invariant: need immediate/register at stack top
26       let val = nondet_i64();
27       emission_context.stack.push(Val::I64(val));
28       emission_context.i64_binop(&mut masm, |masm, dst, src, size| {
29         masm.cmp_with_set(writable!(dst), src, kind, size)?;
30         Ok(TypedReg::i32(dst)) // Return value for cmp is an 'i32'.
31       })
32     },
33   };
34   // check results
35   assert(res.is_ok());
36   emission_context.stack.peek().expect("value at stack top");
37   emission_context.stack.pop().unwrap().is_i32_const();
38   assert(emission_context.stack.inner().is_empty());
39 }

```

Fig. 7: Unit proof for all comparison operations in the MACRO ASSEMBLER.

satisfy invariants that the SUT expects from upstream and external components. When such invariants are not documented, we must discover them during the verification process.

We now present a concrete example of unit proof for the VISITOR functions that handle comparison operations. At a high level, a comparison operation will pop two values from a stack, perform the comparison and push a result to the stack. We would like to prove that all visitor functions for comparison operations are panic free. Therefore, we need to setup the proof environment, choose a comparison kind, pick an operation width and check the returned results. This is the proof we show in Fig. 7.

Setting up the proof environment involves creating a CODEGEN CONTEXT and a MACRO ASSEMBLER (lines 3–6). CODEGEN CONTEXT is parametrized by two phases: PROLOGUE and EMISSION. These are responsible for setting up the function environment (frame) and emitting machine code respectively. We get a CODEGEN CONTEXT in the PROLOGUE phase by invoking `setup_context` using the default configuration for Virtual Machine (VM) offsets⁵. Since we are interested in

⁵VM offsets are pointers to the host environment that a WASM program needs

end-to-end compilation of comparison operations, we change the context to be in the EMISSION phase. Finally, we create a MACRO ASSEMBLER using `setup_masm`. Once this environment has been setup, we can start proving the panic freedom of the system under test—a process that teaches us what invariants are preserved by WINCH. For example, the top value on the stack needs to represent a constant or a register. Moreover, the second value on the stack needs to be the destination register. We satisfy these invariants with lines 17–18, 26–27 and lines 9–10 respectively.

We have encapsulated the `setup_context` and `setup_masm` functions in a separate module for reusability across unit proofs. Their definitions are expanded in Fig. 8. The `setup_context` function takes the VM offsets and produces a CODEGEN CONTEXT in the PROLOGUE phase. It has the responsibility of creating a stack, a frame and a register allocator. We use a function signature with no parameters or returns, but any function signature can be used for the setup routine. The `setup_masm` function creates a MACRO ASSEMBLER for a specific ISA and shared flags specified in Wasmtime. This function has an important result that we discuss in Section V-B.

With the creation of a proof environment that satisfies the WINCH invariants, we continue with the unit proof in Fig. 7. The next step is to choose a comparison kind as shown in line 11. We use the `nondet_u8` function to generate a symbolic 8-bit unsigned integer⁶. By using the generated value, we are effectively choosing a comparison kind non-deterministically. Notice that the same approach is used when selecting the operation width to use (See line 13).

After the comparison functions have been called, we add assertions as shown in lines 35–38. For example, we assert that the result does not report any errors. We also assert that the stack has only one item which is a 32-bit constant. SEABMC returns *unsat* for this proof, giving us confidence that the 16 VISITOR functions that our proof covers are panic free and handle the stack correctly. In this example, we explore all reachable states because there are no unbounded loops to unroll.

B. Discovering Invariants with Executable Counter Examples

To better explain what we mean by executable counterexamples, we use our proof for the panic freedom of the MACRO ASSEMBLER’s constructor. It is shown in the `setup_masm` function of Fig. 8. More specifically, we will explain why the proof needs a concrete value for the pointer size `ptr_size` that is passed into the constructor (See line 35).

⁶For details, see Section V-B

```

#[no_mangle]
pub fn setup_context<'a>(vmoffsets: &'a VMOffsets) ->
    CodeGenContext<'a, Prologue> {
    // stack setup
    let stack = Stack::new();
    // standard frame setup with 0 parameters/returns
    let sig = WasmFuncType::new(
        NoResizableVec::<WasmValType>::new(0),
        NoResizableVec::<WasmValType>::new(0)
    );
    let abi_sig = wasm_sig::<X64ABI>(&sig, 0, 0);
    let locals = DefinedLocals::new::<X64ABI>();
    let frame = Frame::new::<X64ABI>(&abi_sig, &locals.unwrap()).unwrap();
    // setup register allocator
    let gpr = RegBitSet::int(
        ALL_GPR.into(),
        NON_ALLOCATABLE_GPR.into(),
        usize::try_from(MAX_GPR).unwrap(),
    );
    let fpr = RegBitSet::float(
        ALL_FPR.into(),
        NON_ALLOCATABLE_FPR.into(),
        usize::try_from(MAX_FPR).unwrap(),
    );
    let regalloc = regalloc::RegAlloc::from(gpr, fpr);
    // CodeGen in Prologue phase
    let mut ctx = CodeGenContext::new(regalloc, stack, frame, &vmoffsets);
    return ctx;
}

#[no_mangle]
pub fn setup_masm() -> MacroAssembler {
    let isa_flags = cranelift_codegen::x64_settings::Flags::new();
    let shared_flags = cranelift_codegen::settings::Flags::new();
    // invariant: ptr_size has to be equal to 8
    let ptr_size = 8;
    let masm_64 = MacroAssembler::new(ptr_size, shared_flags, isa_flags);
    return masm_64.unwrap();
}

```

Fig. 8: Proof Environment Setup: setup_context and setup_masm.

We had initially defined `ptr_size` with the result of a function `nondet_u8` that takes no arguments and returns a symbolic value. This function is simply a wrapper around a function that SEAHORN understands for symbolic value generation, i.e. `__VERIFIER_nondet_u8`. Under this setup, SEABMC returns SAT, indicating that it finds a path in `setup_masm` that leads to a panic.

The counterexample (cex) pipeline in SEAHORN automatically extracts the values needed to reach the panic and generates the LLVM-IR file in Fig. 9. Lines 2–4 show that SEAHORN expects to provide two symbolic values. It maintains a global value that functions as an index into the array that contains the symbolic values. Using this setup, `__VERIFIER_nondet_u8` will read the index value into `%0`, increment and store it for future use, and return the symbolic value at index `%0`. This way, it returns the first symbolic value the first time it is called. And the second symbolic value the second time it is called.

Our scripts link this generated harness with the original input file to SEABMC and the SEAHORN runtime library to produce a standalone executable. This is the artifact that we call an executable counterexample. It provides a concrete, reproducible witness to the failure and can be run under standard debuggers, bridging the gap between symbolic verification and actionable diagnostics.

With this executable counterexample, the precise sequence of calls required to reach the panic are readily available. Stepping through the counterexample in `rust-lldb` leads us to a panic that encodes an important requirement: the compiler

```

// lines inlined or removed for simplicity
@0 = private constant [2 x i8] c"\00\F7"
@1 = private global i32 0
@__seahorn_cex_count = constant i32 2
define i8 @__VERIFIER_nondet_u8() {
entry:
    %0 = load i32, i32* @1, align 4
    %1 = add i32 %0, 1
    store i32 %1, i32* @1, align 4
    %2 = getelementptr inbounds [2 x i8], [2 x i8]* @0, i32 0, i32 %0
    %3 = load i8, i8* %2, align 1
    ret i8 %3
}

```

Fig. 9: Generated counter-example harness for `nondet_u8`.

expects to run on a 64-bit architecture. We document the invariant and use a concrete value to continue the proof.

C. Results and Takeaways

We have written 14 unit proofs, over three person months, meaningfully verifying 3307 LOC. Our proofs are split into four categories: `visit_setup`, `visit_cmp_ops`, `visit_arith` and `visit_funcs`. The categories are ordered in terms of difficulty. `visit_setup` covers parts of the ASSEMBLER and MACRO ASSEMBLER modules. `visit_cmp_ops` and `visit_arith` cover more parts of the ASSEMBLER and MACRO ASSEMBLER, in addition to parts of the CODEGEN CONTEXT module. Finally, `visit_funcs` covers parts of the CODEGEN CONTEXT that deal with complex data structures and their downstream components.

We show the results of our verification effort in Table I. For each proof category, we show the number of proofs in the collection, effort in man weeks, wall time and verification time in seconds. The `visit_setup` collection has two proofs that take less than one second to run. It took a person week to get the ASSEMBLER and MACRO ASSEMBLER to compile to LLVM 14 and get interesting verification results from SEABMC. The `visit_arith` and `visit_cmp_ops` proof collections have 10 proofs that cover 25 arithmetic and comparison VISITOR functions. These proof collections take less than 15s and 69s respectively. It took two person weeks to prototype verification strategies and leverage our learning to cover the VISITOR functions. The main challenge at this stage was maintaining a stack and a calling context for each proof. The `visit_funcs` collection has two proofs that cover the function signature and call sequence generation for locally defined functions. It took three person weeks to build the right proof environment and ensure proper handling of complex data structures. Despite the high human effort, the proofs take less than 6s to run.

It is important to note the difference in wall time and verification time in Table I. The wall time column represents the time it takes SEAHORN to process LLVM-IR, generate SEA-IR and run SEABMC. The verification time column is a subset of wall time that does not include input preprocessing. Note that most of the time reported for our proofs is not in the verification time column. Instead, a majority of the analysis time is spent simplify the verification problem by running preprocessing, optimization, and static analysis passes on LLVM bitcode. For example, in the case of `visit_arith`, there are 9 proofs that are checked by SEABMC. Out of the total time on these proofs, a fifth is spent on the first

	Proofs	Effort (person-weeks)	Wall Time (s)	Verification Time (s)
visit_setup	2	1	0.87	0.01
visit_cmp_ops	1	2	14.61	1.69
visit_arith	9	2	68.41	11.80
visit_funcs	2	3	5.97	0.27

TABLE I: Verification time for Winch proofs.

preprocessing step with LLVM passes. This illustrates how useful leveraging LLVM passes is for the simplification of the verification conditions generated by SEABMC. We believe this is a worthwhile tradeoff since it is likely to be an important contributor to SEAHORN’s 10x advantage over Kani, as identified in Section IV.

It is noteworthy that the effort required for each proof collection increases linearly. This is a deliberate approach to make progress incrementally and manage the risk of investing time in untractable proof goals. Concretely, we start at the leafs of a visitor function’s call tree and write proofs to establish the callee properties. Doing so iteratively for each caller/callee in the call tree maintains verification momentum without sacrificing the final outcome. This allows us to have more confidence about the panic freedom of WINCH, meaningfully verify more LOC and write more complex proofs after each time unit invested. We recommend this approach since it helped us build our understanding of the codebase incrementally while getting meaningful verification results.

The `visit_funcs` collection of proofs was the most time consuming for two main reasons. First, we had to find intricate invariants that need to be established by upstream components. The executable counterexamples generated by SEAHORN were instrumental to this effort. We found that WINCH requires parameters in a function signature to be stored in specific registers and stack offsets. For example, in our proof for functions with 8 parameters, 6 are put into specific registers and 2 are spilled to the stack, as required by the default WINCH calling conventions. We also learnt that the WINCH ABI uses special locals⁷ that are spilled and treated as parameters. This means that we must reserve enough space for the parameters before invoking the SUT. Second, it is hard for SEAHORN to symbolically reason about HashMaps and HashSets. Therefore, we needed to place stubs for functionality that depended on HashMaps and HashSets.

VI. FUTURE DIRECTION

Our previous study [21] has shown that ownership in LLVM like representation can be used to generate efficient VC for BMC. SEABMC currently does not utilize ownership for VCGEN because all ownership information is lost when compiling Rust to LLVM. This is expected since LLVM does not have ownership concepts. We are developing a pipeline, called SEAURCHIN that extends LLVM with ownership semantics, and compiles Rust to this extended LLVM. With this effort we will retain the advantages of verifying a low-level, close to executable program representation. Simultaneously

verification efficiency will improve by utilizing ownership semantics present in Rust programs.

For benchmarking, the number of unit proofs in VERIFY-RUST can be extended beyond the present number. In addition, VERIFIER-LIB can be extended to support other quality assurance methods (e.g., symbolic execution and fuzzing) to work with the same unit proof. This will be similar in spirit to [1] where multiple tools were benchmarked on the same unit proof.

To verify WINCH, we focused our efforts on a snapshot of the compiler. This results in a separate code-base that needs to be manually kept in sync with the main WINCH repository. Developing a user interface that makes it easy to integrate and maintain unit proofs in the main code-base is an interesting future challenge. This would be a tremendous contribution towards making the use of formal verification tools part of standard engineering practice.

VII. RELATED WORK

Verus [22], Prusti [23], Creusot [24], and Aenas [25] are deductive verifiers for Rust. These deductive tools can model complicated features of the language, like polymorphism, directly. This paper focuses on automatic verification of low-level memory manipulating programs.

The tool closest to SEABMC is KANI. KANI operates on Rust MIR programs, a higher level representation than LLVM IR. For KANI, there is an ever-present opportunity to leverage Rust high level information (e.g., ownership) in generating VC. For SEABMC to benefit similarly, LLVM IR has to be extended with semantics that capture high level information.

The SV-COMP [26] benchmark only considers tools that verify C programs. The benchmarks used in [9] are micro-benchmarks. There are no standardized benchmarks for Rust that we know of. The case studies in our work are a step towards filling this gap.

VIII. CONCLUSION

We show that SEABMC is fast, scalable and practical for bounded model checking of Rust code through two case studies. The first case study contributes 83 unit proofs for SMALLVEC, TINYVEC, SEAVEC, and RESULT-TYPE from the Rust standard library. On these benchmarks, SEABMC is faster than KANI by a factor of 10 or more. It is also more expressive in handling panic behaviour. The second case study contributes a methodology for verifying panic freedom in core parts of WINCH, an industrial WASM-to-executable compiler for various architectures. SEABMC is used to automatically check a collection of 14 proofs that meaningfully verify roughly 3,300 LOC. Together, these case studies demonstrate our claims about SEABMC. SEAHORN is publicly available at <https://github.com/seahorn/seahorn>. The VERIFY-RUST case study is at <https://github.com/seahorn/verify-rust>. The WINCH case study is at <https://github.com/jetafese/winich>.

⁷VM Context pointers to the callee and caller

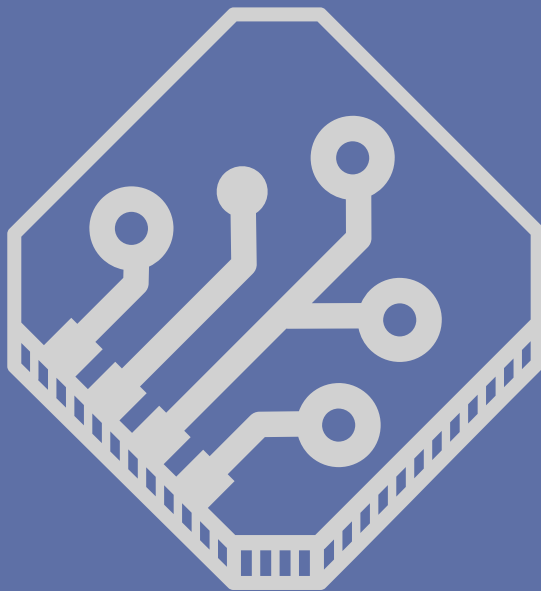
IX. ACKNOWLEDGMENTS

We thank Adrian Jendo, Boris Jancic, Thomas Hart, and Liang Li for their contributions to the VERIFY-RUST benchmarks as part of their undergraduate research assistance-ships at the University of Waterloo.

REFERENCES

- [1] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, “Verifying verified code,” *Innov. Syst. Softw. Eng.*, vol. 18, no. 3, pp. 335–346, 2022. [Online]. Available: <https://doi.org/10.1007/s11334-022-00443-9>
- [2] K. Developers. (2023) The kani book. [Online]. Available: <https://model-checking.github.io/kani/>
- [3] 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>
- [4] “Kani Rust Verifier Blog — model-checking.github.io,” <https://model-checking.github.io/kani-verifier-blog/>, [Accessed 06-05-2025].
- [5] J. Tafese, I. Garcia-Contreras, and A. Gurfinkel, “Btor2MLIR: A Format and Toolchain for Hardware Verification,” in *Formal Methods in Computer Aided Design, FMCAD 2023*, 2023, p. 332.
- [6] J. Tafese and A. Gurfinkel, “Efficient simulation for hardware model checking,” in *Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, N. Bjørner, M. Heule, and A. Voronkov, Eds., vol. 100. EasyChair, 2024, pp. 136–146. [Online]. Available: [/publications/paper/FDRF](https://publications/paper/FDRF)
- [7] 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
- [8] 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.
- [9] M. S. Baranowski, S. He, and Z. Rakamaric, “Verifying rust programs with SMACK,” in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 11138. Springer, 2018, pp. 528–535. [Online]. Available: https://doi.org/10.1007/978-3-030-01090-4_32
- [10] “GitHub - project-oak/rust-verification-tools: RVT is a collection of tools/libraries to support both static and dynamic verification of Rust programs. — github.com,” <https://github.com/project-oak/rust-verification-tools>, [Accessed 08-05-2025].
- [11] “FatLTO; LLVM documentation — llvm.org,” <http://llvm.org/docs/FatLTO.html>, [Accessed 02-05-2025].
- [12] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, “Code-level model checking in the software development workflow at amazon web services,” *Softw. Pract. Exp.*, vol. 51, no. 4, pp. 772–797, 2021. [Online]. Available: <https://doi.org/10.1002/spe.2949>
- [13] “Symbol Mangling - The rustc book — doc.rust-lang.org,” <https://doc.rust-lang.org/rustc/symbol-mangling/index.html>, [Accessed 01-05-2025].
- [14] V. Karakasis, T. Manitaras, V. H. Rusu, R. Sarmiento-Pérez, C. Bignamini, M. Kraushaar, A. Jocksch, S. Omlin, G. Peretti-Pezzi, J. P. S. C. Augusto, B. Friesen, Y. He, L. Gerhardt, B. Cook, Z.-Q. You, S. Khuviz, and K. Tomko, “Enabling continuous testing of HPC systems using Re-Frame,” in *Tools and Techniques for High Performance Computing*, ser. Communications in Computer and Information Science, G. Juckeland and S. Chandrasekaran, Eds., vol. 1190. Cham, Switzerland: Springer International Publishing, Mar. 2020, pp. 49–68.
- [15] Bytecode Alliance, “Wasmtime,” <https://github.com/bytecodealliance/wasmtime/tree/50307096>.
- [16] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, “Fast and secure global payments with Stellar,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, Oct. 2019, pp. 80–96.
- [17] wasmi-labs, “Wasmi,” <https://github.com/wasmi-labs/wasmi/tree/v0.38.0>.
- [18] “WebAssembly Core Specification.” [Online]. Available: <https://www.w3.org/TR/wasm-core-1/>
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Abstraction and Reuse of Object-Oriented Design,” in *ECOOP’93 — Object-Oriented Programming*, G. Goos, J. Hartmanis, and O. M. Nierstrasz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, vol. 707, pp. 406–431.
- [20] Bytecode Alliance, “wasm-parser,” <https://crates.io/crates/wasm-parser>.
- [21] S. Priya and A. Gurfinkel, “Ownership in low-level intermediate representation,” in *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, N. Narodytska and P. Rümmer, Eds. IEEE, 2024, pp. 292–300. [Online]. Available: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_35
- [22] 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>
- [23] 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
- [24] 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
- [25] “Aeneas, a Verification Framework for Rust — aeneasverif.github.io,” <https://aeneasverif.github.io/>, [Accessed 08-05-2025].
- [26] D. Beyer, “State of the art in software verification and witness validation: SV-COMP 2024,” in *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Kovács, Eds., vol. 14572. Springer, 2024, pp. 299–329. [Online]. Available: https://doi.org/10.1007/978-3-031-57256-2_15

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 978-3-85448-084-6



9 783854 480846

www.tuwien.at/academicpress